Ambulance Victoria

# Project Icon development frameworks, standards and practices

**Version 1.0**

**9/12/2019**

**Document Number**: 1

**Document Name**: Project Icon development frameworks, standards and practices

**Document Author:** Sunny Liu

Ambulance Victoria

## Document Control

| Criteria | Details |
|---|---|
| Document ID: | |
| Document title: | Project Icon development frameworks, standards and practices |
| Document owner: | Krishna Nadimpalli |
| Document author: | Sunny Liu |
| Issue date: | Tuesday, 5 November 2019 |

### Version control

| Version | Date | Description | Updated by |
|---|---|---|---|
| 0.1 | 05/11/2019 | First draft | Krishna Nadimpalli |
| 0.2 | 17/11/2019 | Updated document based on feedback provided by ISDA members | Krishna Nadimpalli |
| 1.0 | 9/12/2019 | ISDA endorsed version | Krishna Nadimpalli |
| 1.1 | TBC | Document updated with machine learning coding standards and practices | Abigail Low |
| 1.2 | TBC | Document updated with web application (Node.js, React) coding standards and practices | Mark Geels |

### Document approval

This document requires the following approval:

| Approval Needed | Action | Name / Forum | Date |
|---|---|---|---|
| Baseline Endorsement | Endorse | ISDA | 19 November 2019 |
| Approval of Subsequent Changes | Endorse | ISDA | |
| | | | |

### Linked Documents

This document is linked and should be read in conjunction with the following documents:

| Document Name | Version | Author | Approval Date |
|---|---|---|---|
| Approved Solution Architecture | V1 | KPMG | 28th October 2019 |
| ICT Business Service Catalogue | V1 | Ambulance Victoria | 08 August 2018 |
| Quality Management Plan | V1 | KPMG | TBC |
| Support Design and Service Management Plan | V1 | KPMG | TBC |
| Change and Communications Design | V1 | KPMG | TBC |
| Test Strategy and Test Plans | V1 | KPMG | TBC |

# Table of contents

# 1. Introduction

## 1.1 Purpose of this document

The purpose of this document is to provide guidance to developers on the continuous integration, continuous development and continuous deployment standards, frameworks and practices on AV's Data & Analytics Platform (ADAP). In addition, the document also highlights the minimum expectation from developers in building, testing and deploying code onto the ADAP (e.g. naming conventions, micro-service architecture and key design patterns).

## 1.2 Scope

The scope of DevOps and associated practices including Continuous Integration, Continuous Deployment and Continuous Assurance on the ADAP are a joint responsibility between AV and KPMG. Given that the Agile way of working and the DevOps culture is new to AV, it is expected that the change management and embedding of the practices listed in this document is the responsibility of AV with assistance provided to KPMG (as required).

<u>In scope</u>

The scope of the Development frameworks, standard and practices document is as follows:

- Limited to the AV Data & Analytics Platform
- DevOps roles and governance structure for the ADAP
- Machine learning process, practices and coding standards
- Web application development process, practices and coding standards

<u>Out of scope</u>

- Features and functionality of Azure DevOps. Please refer to Microsoft documentation and training materials (https://azure.microsoft.com/en-au/services/devops/)
- **Infrastructure components** including Active directory, networking, subscriptions, resource groups and DevOps management is **not in scope** and is not covered as part of this document
- Business requirements, business engagement and demand management process
- Incident, support and operations management process
- Project Icon solution architecture and components
- Test strategy and test approach

## 1.3 Audience

The audience for this document include:

- Developers from KPMG, AV and external third parties
- KPMG development administrators
- AV Infrastructure
- AV Network
- AV Security

# 2. DevOps and Agile Development at AV

## 2.1 Overview

AV is building a data and analytics platform (ADAP) on Azure to address the challenges identified and prepare AV for the future. ADAP is designed to be **a highly available, scalable and resilient next-generation data and analytics platform** that can integrate with the range of current and anticipated data sources in order to enable paramedics and their support crew to make real-time and predictive decisions to improve operational performance and clinical outcomes.

As part of the transition to the ADAP, AV is also keen to **transition the organization into an Agile/DevOps mindset** and ways of working. Agile focuses on improving the speed to value to Customers and end users for development activities via continuous iterative development and small releases. The speed to value however is dependent on a supporting operating model which includes the alignment of people, process and technology that enables continuous delivery.

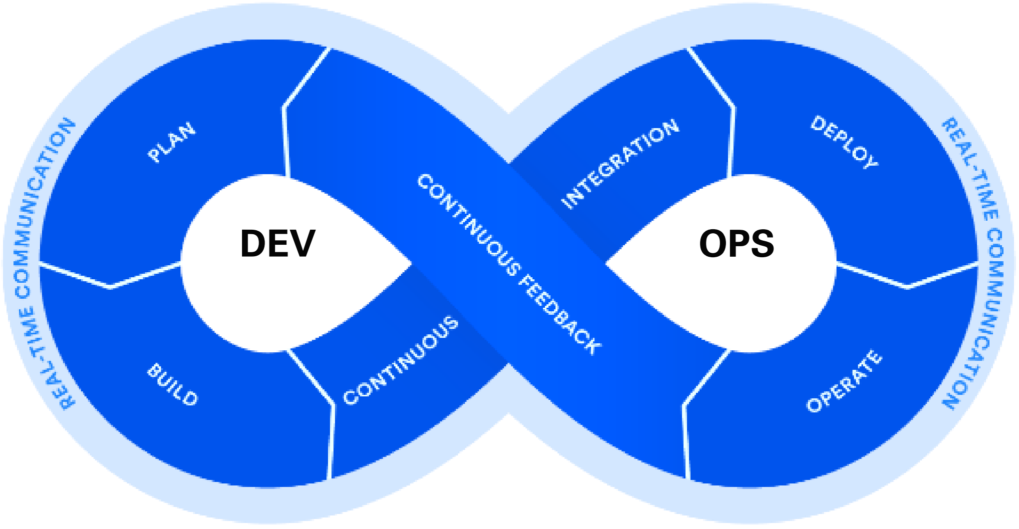
From AV's perspective, the **Agile way of working** and the **continuous integration / continuous delivery** approach is a significant shift from the prevailing waterfall development culture and mindset. The embedding of the new way of working will be a journey for AV and will require significant endorsement and support from project leadership during the project.

## 2.2 DevOps and the Application Lifecycle

DevOps is not a technology solution or a software development methodology but the union of people, process, and products to enable continuous delivery of value to your end users. There is no one true way or prescribed methodology for DevOps and needs to be tailored for Ambulance Victoria.

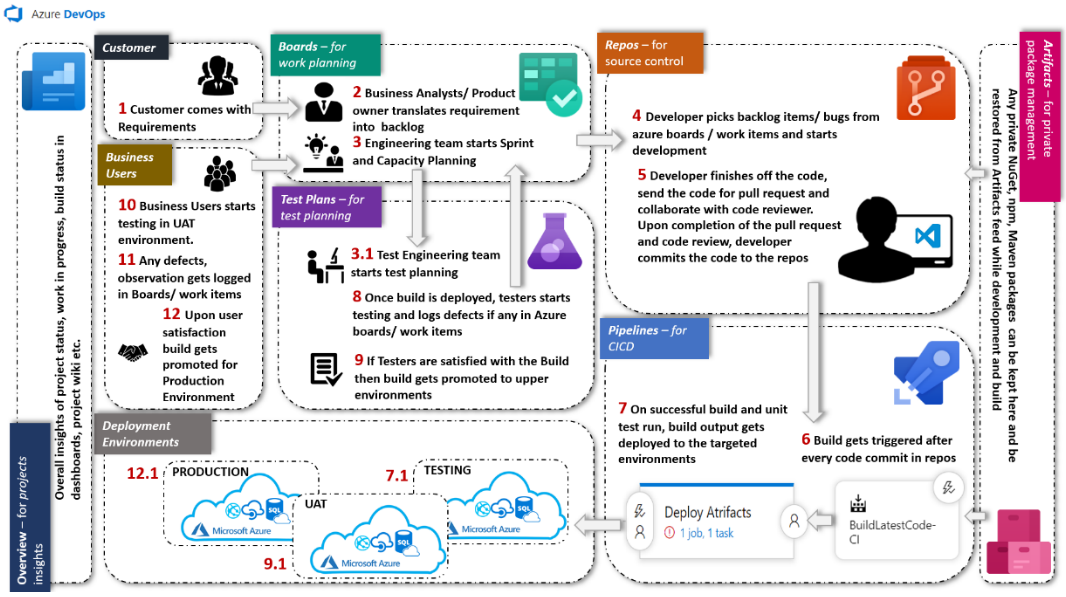
The objective of DevOps is to enable accelerate delivery, while still delivering reliable products and services. This can be achieved by bringing people together, through shared common goals, increased collaboration and a focus on improvement. Technology plays a key role in helping to execute a DevOps strategy by enabling teams to collaborate more, enhance productivity, facilitate experimentation and automate their processes from development, through to delivery and operations. It's also about delivering value faster by increasing efficiency, eliminating waste and streamlining feedback back to developers to continuously improve the product.

The diagram below depicts the relationship between DevOps and the application lifecycle. The remainder of this document is structured around these six key stages of the application lifecycle and the practices expected for each stage during development activities on the ADAP.



## 2.3 High-level end-to-end business process for Agile development

The following diagram depicts the high-level end-to-end business process from defining the business requirements, developing the code and releasing into the environments.



Azure DevOps

**Customer**

1 Customer comes with Requirements

**Boards – for work planning**

2 Business Analysts/ Product owner translates requirement into backlog

3 Engineering team starts Sprint and Capacity Planning

**Repos – for source control**

4 Developer picks backlog items/ bugs from azure boards / work items and starts development

5 Developer finishes off the code, send the code for pull request and collaborate with code reviewer. Upon completion of the pull request and code review, developer commits the code to the repos

**Artifacts – for private package management**

Any private NuGet, npm, Maven packages can be kept here and be restored from Artifacts feed while development and build

**Pipelines – for CICD**

6 Build gets triggered after every code commit in repos

7 On successful build and unit test run, build output gets deployed to the targeted environments

**Test Plans – for test planning**

3.1 Test Engineering team starts test planning

8 Once build is deployed, testers starts testing and logs defects if any in Azure boards/ work items

9 If Testers are satisfied with the Build then build gets promoted to upper environments

**Deployment Environments**

**Business Users**

10 Business Users starts testing in UAT environment.

11 Any defects, observation gets logged in Boards/ work items

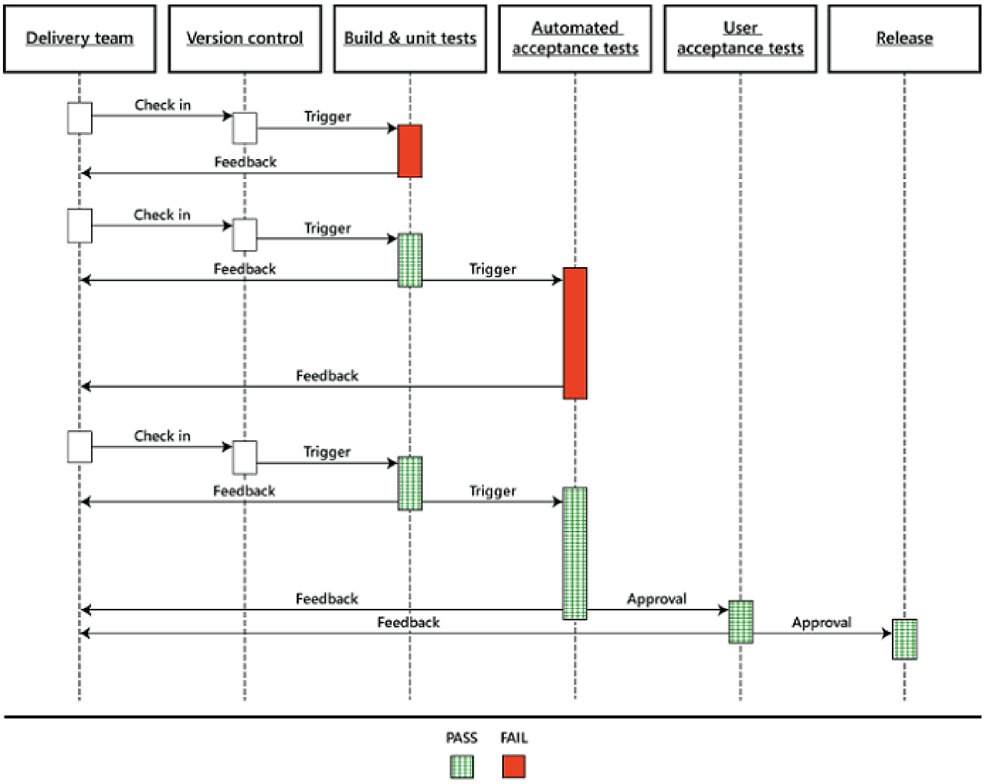
12 Upon user satisfaction build gets promoted for Production Environment

**Overview – for projects insights**

Overall insights of project status, work in progress, build status in dashboards, project wiki etc.

## 2.4 High-level code deployment process

The following diagram provides a high-level overview of the Agile development process and how code is developed by the Delivery team and progresses through the various stages up to Release into the Production environment.



## 2.5 DevOps practices

The table below represents a list of practices that when followed during the development of the ADAP will help embed a DevOps culture at Ambulance Victoria.

| # | Practice | Description |
|---|---|---|
| 1 | Version Control All Production Artifacts | Version control is the practice of managing code in versions – tracking revisions and change history to make code easy to review and recover. This practice is usually implemented using version control systems such as Git which allow multiple developers to collaborate in authoring code. These systems provide a clear process to merge code changes that happen in the same files, handle conflicts and roll back changes to earlier states. The use of version control is a fundamental DevOps practice, helping development teams work together, divide coding tasks between team members and store all code for easy recovery if needed. Version control is also a necessary element in other practices such as continuous integration and infrastructure as code. |
| 2 | Continuous Integration and Deployment | Code should be integrated regularly (minimum daily) into the trunk as opposed to integrating at the end of the release. Developers should not hang onto private code branches that are not integrated into the trunk. |
| 3 | Automated Acceptance Testing | Instil Test Driven Development into the team where there focus is on the execution of continuous testing and automating this testing as much as possible. The focus of acceptance testing should be on successful build and releases. Shift the focus on resolving issues when the build breaks but also when something breaks in an automated user test, an integration test, or a system test. This step keeps code in an always-deployable state. |
| 4 | Peer Review of Production Changes | Utilise peer reviews for better quality; leverage the team's familiarity, shared goals, and mutual accountability, as opposed to external change approval (such as a change advisory board). |
| 5 | High-Trust Culture | The development, operations and broader project team should continuously work towards practices that build a high-trust culture. This is both a practice and an outcome result from a single source of truth, peer reviews, and shared goals |
| 6 | Proactive Monitoring of the Production Environment | Embed monitoring techniques on all components of the ADAP and ensure communication across the teams so everyone can see, understand, and affect end results. |
| 7 | Win-Win Relationship (and Outcomes) between Dev and Ops | This approach counters the learned behavior that deployments hurt. By deploying code into production on shorter timeframes, there is a change in Operations. Deployments don't have to be done at midnight on Friday with Ops working all weekend to get things running. When Ops employees are working the same hours as Dev, there is a sense of teamwork and joint accomplishment. |
| 8 | Prioritise security incident response and resolution | All security incidents or vulnerabilities identified through either code scans or manual tests are to be resolved by the DevOps team as a matter of priority. |

## 2.6 Roles and responsibilities

The following table lists the key roles and responsibilities on the ADAP project from a development perspective.

| Role | Description | Role Assigned to |
|---|---|---|
| Project Administrators | This role is responsible for the administration of the ADAP Project in Azure DevOps. The Administrator is also responsible for the Project structure including Teams and Area Paths. | - KPMG Project Administrator<br>- AV Azure DevOps Organisation Administrators |
| Contributors | Developers on the project who can contribute to repositories, boards and can perform pull requests on protected branches. | - KPMG developers<br>- AV internal developers<br>- External developers |
| Dev Admin | Responsible for the review and approval of code to protected branches. | - KPMG Integration Lead<br>- KPMG Security Lead (Prod release only)<br>- AV Security (Prod release only)<br>- AV Infrastructure (Prod release only) |
| Build Administrators | Responsible for creating the build pipelines for the deployment of code into Azure services | - KPMG Build pipeline creators |
| Tester | Responsible for performing testing activities (both functional and non-functional) on code to be deployed onto the platform | - KPMG Testers<br>- AV Testers |
| Project Manager | Responsible for business engagement and demand management which requires use of Azure Boards | - KPMG Business Engagement and Demand Manager |

## 2.7 Third party development on ADAP

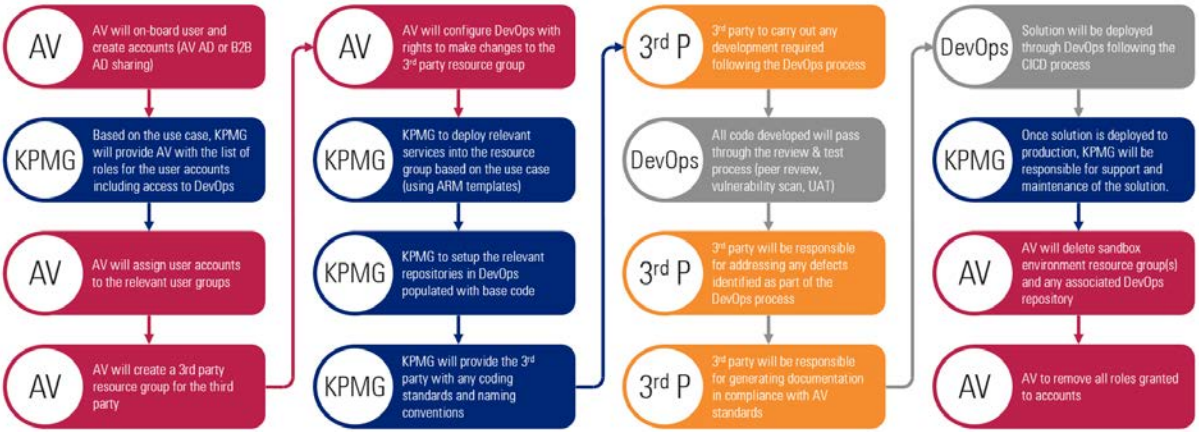
KPMG will be responsible for the maintenance and support of the ADAP platform. All code developed by non-KPMG parties (Contributor) will go through a review and approval process by KPMG (Dev Admin role) in accordance with the standards documented below.

In terms of access, all third party developers will receive the “contributor” role. Third party developers will not have any permissions to modify any builds, pipelines or releases.

It is assumed that

- third party developers will not use any Azure service not already included in the current architecture design
- If any additional services are required, change requests will need to be submitted as per the governance structure for infrastructure deployment. Security review and risk modelling will be carried out to determine if additional security controls are require

The following diagram depicts the high level end-to-end process for development activities by third party developers:


AV
AV will on-board user and create accounts (AV AD or B2B AD sharing)
KPMG
Based on the use case, KPMG will provide AV with the list of roles for the user accounts including access to DevOps
AV
AV will assign user accounts to the relevant user groups
AV
AV will create a 3rd party resource group for the third party
AV
AV will configure DevOps with rights to make changes to the 3rd party resource group
KPMG
KPMG to deploy relevant services into the resource group based on the use case (using ARM templates)
KPMG
KPMG to setup the relevant repositories in DevOps populated with base code
KPMG
KPMG will provide the 3rd party with any coding standards and naming conventions
3rd P
3rd party to carry out any development required following the DevOps process
DevOps
All code developed will pass through the review & test process (peer review, vulnerability scan, UAT)
3rd P
3rd party will be responsible for addressing any defects identified as part of the DevOps process
3rd P
3rd party will be responsible for generating documentation in compliance with AV standards
DevOps
Solution will be deployed through DevOps following the CICD process
KPMG
Once solution is deployed to production, KPMG will be responsible for support and maintenance of the solution.
AV
AV will delete sandbox environment resource group(s) and any associated DevOps repository
AV
AV to remove all roles granted to accounts

# 3. Azure DevOps (Technology platform)

## 3.1 Overview

**Azure DevOps was chosen as the technology platform** to support the embedding of DevOps practices and ways of working.

It provides a suite of tools for all aspects of the Agile methodology including:

- **Boards** for tracking work items, boards, backlog and Sprints using the Agile methodology
- **Repos** for repository management (code commits, pushes, branches and pull requests)
- **Pipelines** for Continuous Integration and Continuous Development / Continuous Deployment builds and releases
- **Test Plans** for defining, publishing and running unit, system and regression test and
- **Artefacts** for package repository management.

Further information regarding Azure DevOps can be found at: https://docs.microsoft.com/en-us/azure/devops/?view=azure-devops

## 3.2 Roles & Permissions

Azure DevOps provides the capability to define granular role-based access control and management. For Project Icon, Team members will be assigned to an Azure AD user group with their AV or linked B2B account which will correspond to a specific role as defined in section 2.4. These roles will dictate what each user have access to and what changes can they make on Azure DevOps. The principle of least access is applied when assigning RBAC to user groups. In addition, segregation of duties for the roles identified will be implemented to ensure that no conflicting rights are assigned to any given user.

Role permissions proposed below will be discussed and agreed between KPMG and AV. KPMG will be responsible for defining the roles and responsibilities below to individual user and user groups with the DevOps project.

The following table provides the roles related to Project Icon and their associated permissions on Azure DevOps. While every effort has been taken to utilise out-of-the-box roles and permissions, additional roles have been defined to cater for the nature of the project.

| Action | | ROLES | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Project Administrators | Contributors | Release Administrators | Build Administrators | Tester | Project Manager | Reader |
| | | General | | | | | | |
| | Delete team project | ✓ | | | | | | |
| | Edit project-level information | ✓ | | | | | | |
| | Manage project properties | ✓ | | | | | | |
| | Rename team project | ✓ | | | | | | |
| | Suppress notification for work item updates | ✓ | | | | | | |

| Action | | ROLES | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Project Administrators | Contributors | Release Administrators | Build Administrators | Tester | Project Manager | Reader |
| | Update project visibility | ✓ | | | | | | |
| | View project-level information | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **Boards** | | | | | | | | |
| | Bypass rules on work item updates | ✓ | | | | | | |
| | Change process on team project | ✓ | | | | | ✓ | |
| | Create tag definition | ✓ | | | | | ✓ | |
| | Delete and restore work items | ✓ | | | | | ✓ | |
| | Move work items out of this project | ✓ | | | | | ✓ | |
| | Permanently delete work items | ✓ | | | | | ✓ | |
| **Boards Area** | | | | | | | | |
| | Create child nodes | ✓ | | | | | ✓ | |
| | Delete this node | ✓ | | | | | ✓ | |
| | Edit this node | ✓ | | | | | ✓ | |
| | Edit work items in this node | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | View permissions for this node | ✓ | | | | | | |
| | View work items in this node | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **Analytics** | | | | | | | | |
| | Delete shared analytics views | ✓ | | | | | ✓ | |
| | Edit shared analytics views | ✓ | | | | | ✓ | |

| Action | | ROLES | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Project Administrators | Contributors | Release Administrators | Build Administrators | Tester | Project Manager | Reader |
| | view analytics | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | **Test Plans** | | | | | | | |
| | Create test runs | ✓ | ✓ | | | ✓ | | |
| | Delete test runs | ✓ | | | | | | |
| | Manage test configuration | ✓ | ✓ | | | ✓ | | |
| | Manage test environments | ✓ | ✓ | | | ✓ | | |
| | View test runs | ✓ | ✓ | | | ✓ | | |
| | **Repository** | | | | | | | |
| | Bypass policies when completing pull requests | ✓ | | ✓ | | | | |
| | Bypass policies when pushing | ✓ | | ✓ | | | | |
| | Contribute | ✓ | ✓ | | | | | |
| | Contribute to pull request | ✓ | ✓ | ✓ | | | | |
| | Create branch | ✓ | ✓ | ✓ | | | | |
| | Create repository | ✓ | ✓ | ✓ | | | | |
| | Create tag | ✓ | ✓ | ✓ | | | | |
| | Delete repository | ✓ | | | | | | |
| | Edit policies | ✓ | | ✓ | | | | |
| | Force push | ✓ | | ✓ | | | | |
| | Manage notes | ✓ | | ✓ | | | | |

| Action | | ROLES | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Project Administrators | Contributors | Release Administrators | Build Administrators | Tester | Project Manager | Reader |
| | Manage permissions | ✓ | | | | | | |
| | Read | ✓ | ✓ | ✓ | | | | |
| | Remove others' locks | ✓ | | ✓ | | | | |
| | Rename repository | ✓ | | | | | | |
| **Build** | | | | | | | | |
| | Administer build permissions | ✓ | | | | | | |
| | Delete build pipeline | ✓ | | ✓ | ✓ | | | |
| | Delete builds | ✓ | | ✓ | ✓ | | | |
| | Destroy builds | ✓ | | ✓ | ✓ | | | |
| | Edit build pipeline | ✓ | | ✓ | ✓ | | | |
| | Edit build quality | ✓ | | ✓ | ✓ | | | |
| | Manage build quality | ✓ | | ✓ | ✓ | | | |
| | Manage build queue | ✓ | | | ✓ | | | |
| | Override check-in validation by build | ✓ | | | ✓ | | | |
| | Queue builds | ✓ | | | ✓ | | | |
| | Retain indefinitely | ✓ | | | ✓ | | | |
| | Stop builds | ✓ | | | ✓ | | | |
| | Update build information | ✓ | | | ✓ | | | |
| | View build pipeline | ✓ | ✓ | ✓ | ✓ | | | |

| Action | | ROLES | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Project Administrators | Contributors | Release Administrators | Build Administrators | Tester | Project Manager | Reader |
| | View builds | ✓ | ✓ | ✓ | ✓ | | | |
| **Release** | | | | | | | | |
| | Administer release permissions | ✓ | | | | | | |
| | Create releases | ✓ | | ✓ | | | | |
| | Delete release pipeline | ✓ | | ✓ | | | | |
| | Delete release stage | ✓ | | ✓ | | | | |
| | Delete releases | ✓ | | ✓ | | | | |
| | Edit release pipeline | ✓ | | ✓ | | | | |
| | Edit release stage | ✓ | | ✓ | | | | |
| | Manage deployment | ✓ | | ✓ | | | | |
| | Manage release approvers | ✓ | | ✓ | | | | |
| | Manage releases | ✓ | | ✓ | | | | |
| | View release pipelines | ✓ | ✓ | ✓ | ✓ | | | |
| | View releases | ✓ | ✓ | ✓ | ✓ | | | |

## 3.3 Azure DevOps licensing

The following licensing is recommended for the roles identified above:

| Role | Azure DevOps license |
|---|---|
| **Project Administrators** | Basic |
| **Contributors** | Basic + Test Plan |
| **Release Administrators** | Basic |
| **Build Administrators** | Basic |
| **Tester** | Basic + Test Plan |
| **Project Manager** | Basic |
| **Reader** | Stakeholder |

## 3.4 IaaS, PaaS and Code Deployment using Azure DevOps

Azure DevOps will be utilised primarily for PaaS deployment and code deployment on the ADAP. The only IaaS component currently included in the solution architecture is the Data Management Gateway.

One of the key differences in terms of roles and responsibilities for deployment onto the Azure platform relates to whether Azure service or code is being deployed onto the platform. The deployment of either component (PaaS or code) requires a different involvement from AV stakeholders.

### 3.4.1 PaaS components

The following are the key considerations for Platform-as-a-Service deployments onto the ADAP.

- ADAP PaaS components will be deployed onto ADAP using ARM templates
- These ARM templates are stored in a separate repository
- A CI/CD pipeline in Azure DevOps Pipelines will be developed to deploy these services into the relevant environment (DEV, TST, STG, PRD)
- As an example, deployment of an Azure SQL Database will require server level configuration such as Vnet setup, IP whitelisting, and Azure Active Directory Admin.
- The nominated AV team members will approve prior to the deployment of these components into any environment (DEV, TST, STG). Approvers listed below.
- Once the Azure Services are deployed into the environments, any changes to the services are reflected in the relevant ARM template
- The update to the ARM template triggers and subsequent commit will trigger a build and release pipeline which will (again) require approval from nominated AV infrastructure, security and network personnel.
- **Approval for PaaS components will be required from the following members and will be configured into the Branch policies:**
  - AV Infrastructure
  - AV Security
  - KPMG Solution Architect

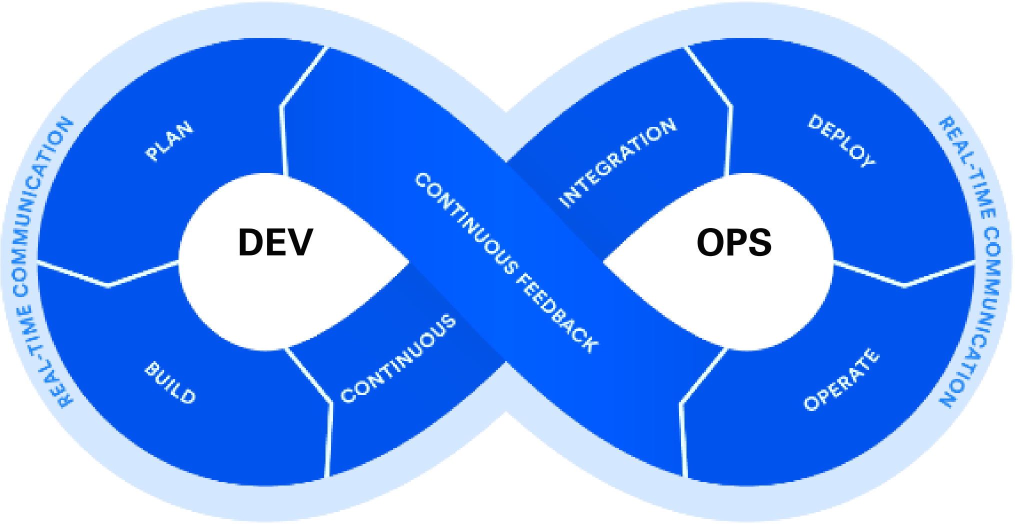
### 3.4.2 Code integration and deployment

The following are the key considerations for code deployment onto the ADAP:

- Code developed on the platform will be deployed to the relevant feature branch in the repository
- Speed is the focus for code integration onto the platform. This is to encourage developers to regularly check-in their code into the main trunk as soon as possible
- **Approvals**
  - Merging of code into the TST and STG environments may require approval from nominated AV and KPMG stakeholders. This needs to occur through a Continuous Assurance (reviewal) process by both KPMG and AV team members. It is the responsibility of the stakeholders to be available for testing and assurance activities during the Sprints.
  - Merging of code into the Production environment requires involvement from AV’s infrastructure, network and security teams

# 4. DevOps standards and practices

The following section detail the standards and practices for each phase of the DevOps cycle including, where possible, the link to Azure DevOps technology components. For reference, the DevOps cycle is provided below.



## 4.1 Phase 1: Plan

Plan is the first step in the DevOps process.  Each sprint must be planned out carefully to ensure that the team is focused and delivering to the right business requirement at the right time.  This section will detail out the phases and process for Plan & Track

### 4.1.1 Business Requirements Gathering

All feature develop or bug resolution must start with business requirements.  The business requirements can be sourced from multiple channels (including Users, AV teams, Incidents, Forums etc) across AV. As part of the DevOps process, all business requirements will be captured as user stories.  A single business requirement may be translated into one or multiple user stories depending upon the size of requirement.  Once the user story is defined, it will be recorded in the product backlog.

Each backlog item will have

- Detailed description of the business requirement.
- An agreed definition of the acceptance criteria with the Product Owner.
- Detailed tasks and sub-tasks generated as the part of the business requirement gathering process
- Agreed priority and activity type of the respective task and sub-task.
- Priority and any risks associated with the user story.

The above steps will be iterated to capture all of the business requirements incrementally. All of the captured requirements will be available in the product backlog for prioritization and sprint planning.

### 4.1.2 Business Requirements Prioritization

The demand manager will work with the product owners to priories each user story prioritised based on its priority and risks.  Where competing priorities exist between user stories, both product owners will be invited to work with the demand manager to identify a suitable priority.

### 4.1.3 Sprint Planning

Once business requirements are gathered, translated into user stories and prioritized, sprint planning will be used to determine the scope of work to be completed within each sprint. Sprint planning is an event in the agile framework where the team determines the product backlog items they will work on during that sprint and discusses their initial plan for completing those product backlog items.

Sprint planning usually occurs either towards the end of the previous Sprint or early in the current Sprint. Depending on the rhythm of the team, timing of this planning activity may be adjusted – e.g. to have sprint planning finished before the commencement of a new sprint. Sprint planning will aim to incorporate all feedbacks from sprint review and retrospective from the previous sprint to increase teamwork and improve sprint delivery capabilities.

Teams may find it helpful to establish a sprint goal and use that as the basis by which they determine which product backlog items they work on during that sprint. Below are the main players and the activities that will contribute to the sprint planning process:

- A “**product owner**” identifies the candidate product backlog items and their relative priorities, as well as proposes a sprint goal.
- The “**team**” members will provide estimate on the effort required to complete each task in the forms of story points.
- A “**demand manager**” will facilitates sprint planning in order to ensure that the discussion is effective and that there is agreement to the sprint goal and that the appropriate tasks are included in the sprint backlog based on the priority of each ticket and the estimated efforts.

### 4.1.4 Azure DevOps using Kanban boards

Once sprint planning is complete, every task will be created on Kanban as a ticket. Feature branches can either be created on the Developer’s local machine and pushed to Azure DevOps or created directly on DevOps repository. It is a requirement that all feature branches are linked to a ticket on the Kanban board. Developer will contribute to the respective feature branch by writing test cases, developing code, testing code and submitting the code to the feature branch. Both manual and automated test cases will be documented in the task description on Kanban. Testers may refer to the task descriptions to perform testing on a particular feature.

Tasks will be closed as they are completed. User stories will be closed if all tasks under the user story is completed. If a user story is open at the end of a sprint, it will be pushed into the next sprint.

## 4.2 Phase 2: Build

Once each sprint has commenced, developers will be assigned tickets to develop a feature or resolve a bug. Because development is designed for the cloud, there are a few software quality pillars and design principles & patterns that must be followed to ensure all code developed accurately functions within the cloud environment. Consistent development standards, naming conventions and repository & branch structure are also needed to ensure high maintainability.

### 4.2.1 Software quality pillars

The following software quality pillars should provide guidance on all code built on the platform. The key considerations around the software quality pillars listed below are included in the design documentation produced as part of each Sprint. This documentation is provided to AV Solution Architects and stakeholders for review.

#### 4.2.1.1 Resiliency

Software resiliency ensures that software can perform at an acceptable service level when problems occur in one or more parts of the system. In order to make the solution resilient, recovery methods are implemented to reduce the amount of failures as well as minimise the impact when a failure occurs. Specifically for ADAP, all code written must be able to effectively deal with issues such as non-responsiveness, latency, incomplete and out-of order messages etc.  Below are some of the approaches & implementation strategies that will be employed on the platform address the issues identified:

- **Asynchronous communication (e.g. between messages) across microservices –** this reduces latency issues in message processing by reducing wait-time between requests
- **Use retries with an exponential back-off –** this is an industry standard technique that retries an operation, with an exponentially increasing wait time, up to a maximum retry count to improve reliability.  Busy network is a common scenario where retry is needed.  The exponential increasing wait time prevents the case where an already busy network is bombard with additional retries by all systems in a short period of time causing additional congestion.
- **Plan around network timeouts –** this increases uptime by avoiding network outages which improves overall reliability
- **Use a circuit breaker pattern –** configurable timeouts and retries will be included in code by developers for fault tolerance. However, a failure due to the communication with another components is likely to be long-lasting, using a retry mechanism can affect the responsiveness of the application. A circuit breaker pattern involves wrapping a fragile function call (typically an integration point with another service) in a special (circuit breaker) object, which monitors for failures. Once the failures reach a certain threshold, the circuit breaker trips, and all further calls to the circuit breaker return with an error, without the protected call being made at all.l
- **Provide fall backs –** contingency plans are implemented within the code to have an alternative solution in the event of a failure.
- **Limit the number of queued requests -** By setting a maximum amount of queued request at any given time, non-responsiveness and latency issues are minimised.

#### 4.2.1.2 Security

Maintaining the security of code is a vital component of providing a high quality solution. ADS Essential 8 and ISM requirements will be adhered to during deployment as per the architecture document.  Within development code, there are a series of principles and methods that can be implemented to improve security. The exact modules and methods used are dependent on the coding language being used, but some general principles can apply to all.

At a high-level, it is the responsibility of the Developer to ensure the security of the code that is cloned from the repositories onto Developer's laptops. Developer's must comply to AV Security standards and policies around data transfer from machines and development on non-AV laptops requires written AV approval.

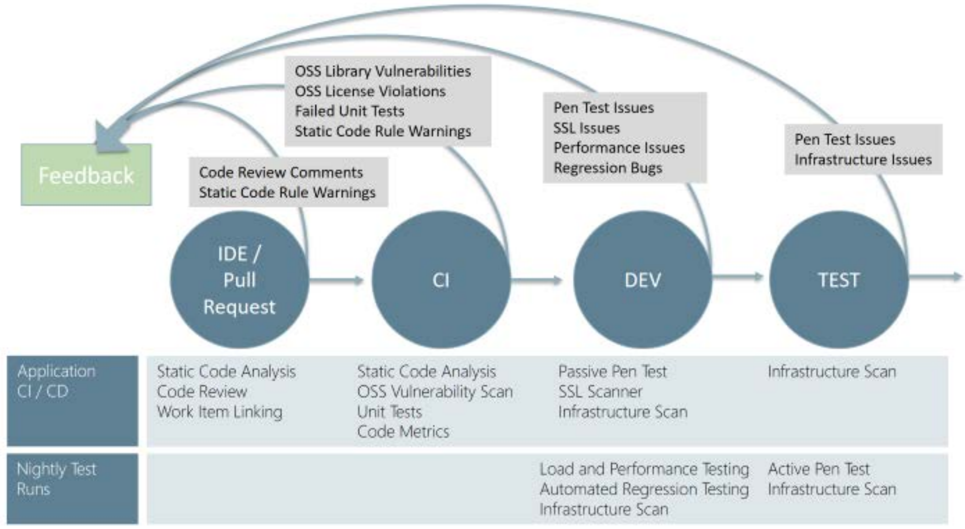
The Azure platform provides tools and capabilities to aid in creating secure solutions. Azure Key Vault is a cloud service used to manage keys, secrets, and certificates. Key Vault eliminates the need for developers to store security information in the code. Key Vault is available directly from the Azure portal.  More information are available via this link.  Azure Key Vault will be integrated within ADAP for all credential management.

Apart from Azure Key Vault, the following coding practices will also be used to improve security:

- **Avoid hard coding any variables (especially log-in credentials) -** hard coding login credentials or any other sensitive data creates vulnerability in the code.  Azure Key Vault will be used to manage credentials.
- **Handle sensitive data using secure methods -** Securing methods through general practices including encapsulation, threat modelling, re-use of known and trusted components and integration of security scans makes code harder to exploit and better protects data being used by functions.
- **Implement data anonymisation –** data containing PII information will be anonymised using a Hash (SHA256) and Salt so that developers only have access to masked data during development and testing. Data ingested into the Platform will be classified during the design

phase of the Sprint with AV stakeholders and the Hash and Salt technique will be applied to anonymise the data. The original PII is not stored on the platform.

The following diagram shows the incorporation of the regular security testing throughout the stages of code development.



| | | | | |
|---|---|---|---|---|
| Application CI / CD | Static Code Analysis<br>Code Review<br>Work Item Linking | Static Code Analysis<br>OSS Vulnerability Scan<br>Unit Tests<br>Code Metrics | Passive Pen Test<br>SSL Scanner<br>Infrastructure Scan | Infrastructure Scan |
| Nightly Test Runs | | | Load and Performance Testing<br>Automated Regression Testing<br>Infrastructure Scan | Active Pen Test<br>Infrastructure Scan |

Microsoft best practices recommend the following to add continuous security validation to CI/CD pipelines[1].

- *Static code analysis* during the CI Build using SonarCloud and Roslyn Security Analyzers[2][3]
- *Code review using* branch policies to ensure manual code reviews of features and functionality that is linked to Work Items
- *OWASP scan* using VSTS extensions[4] which is run during the Release pipeline
- Azure Security Centre and Azure Policies for on-going vulnerability scans on the platform components

Security issues that have been identified through code-reviews will be recorded on the Azure DevOps wiki for reference by the development team. Refers to security issues that are identified during scans performed during Sprint delivery. Any security findings that have been identified through the scan are alerted to KPMG and AV (project) security team who then perform an assessment on the impact and criticality of the finding. There maybe circumstances where the finding is deemed as low by the project team at which point in time it's included in the 'Known security issues register' for future reference. Subsequent scans of the code base will be referenced to the known-issues register on the Azure DevOps wiki. All security related findings from static code analysis or manual code review will require input from AV's Security Team and the project's security resource to agree upon the remediation. The remediation could be inclusion of the issue into the known issues register or remediation of the security issue. All security issues are documented on the Azure DevOps board as Bugs with the classification determined and agreed to between AV's Security Team and the project's security resource.

### 4.2.1.3 Scalability

Scalability of code is the code's capacity to meet an increase in size or scale.  On the ADAP, scalability is primarily dependant on code efficiency (software) and service scalability (hardware). While some scalability challenges may be address by scaling the service to a higher tier or increase

[1] https://docs.microsoft.com/en-us/azure/devops/migrate/security-validation-cicd-pipeline

[2] https://docs.microsoft.com/en-us/azure/security/develop/threat-modeling-tool

[3] https://secdevtools.azurewebsites.net

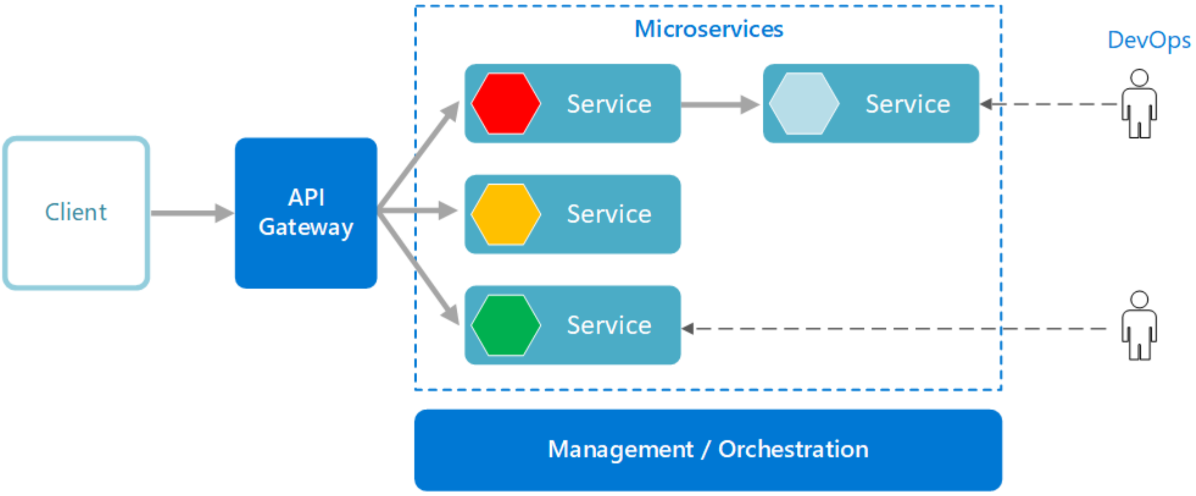
[4] https://github.com/deliveron/owasp-zap-vsts-extension

the number of instances within the service, majority of the scalability gains comes from implementation in the software. Scalable code on the platform needs to adhere to the following principles:

- **Use common utilities where possible -** pre-existing modules for common data access and data movement patterns will be built on the platform. These functions are designed for security, scalability and optimisation. It is expected that developers performing common functions should verify whether a library or utility exists as this will save time and complexity in development activities.
- **Avoid creating functions that have a high time or memory complexity -** Functions that run for either more than 2-5 minutes or require more than 5-7GB of memory will need to be reviewed and assessed for code complexity as such large or complex code bases do not scale appropriately.
- **Use asynchronous calls –** asynchronous calls increases parallelism and reduces blocking between processes. Hence, asynchronous calls will help reduce latency and increase scalability
- **Avoid locking resources –** this helps to reduces latency in the same manner as asynchronous calls
- **Minimize the time that connections and resources are in use –** this directly increases the efficiency of the code which allows for more scaling without the introduction of latency

### 4.2.2 Microservices architecture

Following the principle agreed to in the architecture design for ADAP, microservice architecture will be used for the platform. A microservices architecture consists of a collection of small, autonomous services. Each service is self-contained and should implement a single business capability.  An example implementation is outlined in the picture below.



Microservices architecture is selected for the project for the following reasons: the loose coupling of microservices makes it easier to resolve bugs and update services without redeploying the entire application by minimising dependencies. Microservices are small enough that only a small team of developers are required to build a single feature in a reasonable amount of time. Lastly, this approach allow for scalability of microservices independent of others, allowing the scaling up/out of specific services without having to scale out the entire application.

Therefore, developers should always aim to design functions to support the microservices architecture on the platform.

### 4.2.3 Microservices design principles

There are a number of guiding principles that not only allow developers to achieve an isolated microservice design for each microservice, but still providing a level of consistency across the platform.  These principles provide a general framework to developing microservices without limiting developer's ability to tailor the function to a specific requirement, allowing a balance between consistency and productivity.

- **Single Responsibility Principle (SRP) of Microservices** - Each microservice should be self-contained and have only a single responsibility. This allows for microservices to be relatively independent, more easily worked on by a small team, and relative scalability.
- **Decomposition of Microservices by Business Capability** - Microservices should align directly to a corresponding business capability. This allows for a consistent approach to the logical architecture that best fits this application design and further reinforces the minimisation of dependencies between services.
- **Adherence to Coding Standards** - Microservices must follow the relevant coding standard that has been prescribed in the coding standard section of this document. This includes naming conventions, documentation and other formatting standards such as indentation.
- **Communication of Services Through APIs** - Services should only communicate through well-designed APIs. This keeps microservices self-contained and keeps the communication between services standardized.

### 4.2.4 Key Design Patterns

On top of using microservices architecture, implementing the right software design patterns provide best practice solutions to common software design problems.  Design patterns can also speed up the development process by providing tested, proven development paradigms.  Some design patterns will be derived from the principles descripted in the microservices design principles.

These design patterns are useful for building reliable, scalable and secure applications in the cloud.

| Design Pattern type | Description | High level Implementation detail |
|---|---|---|
| Asynchronous request-reply | Decouple backend processing from a frontend host, where backend processing needs to be asynchronous, but the frontend still needs a clear response. | Frontend and other applications access ADAP APIs asynchronously to get data for respective functionality. |
| Command and Query Responsibility Segregation | Segregate operations that read data from operations that update data by using separate interfaces. | Read and Write operations are separated for simpler designs and better isolation of resource usage. |
| External Configuration Store | Move configuration information out of the application deployment package to a centralized location. This can provide opportunities for easier management and control of configuration data, and for sharing configuration data across applications and application instances. | A NoSQL database is used to store all configuration. Re-use configuration across components, reduce downtime when configuration changes are made and no sprawl of configuration. |
| Federated Identity | Delegate authentication to an external identity provider. This can simplify development, minimize the requirement for user administration, and improve the user experience of the application. | The Federated Access is implemented in Azure Active Directory. Multiple applications get access from one identity provider. |
| Pipes and Filters | Decompose a task that performs complex processing into a series of separate elements that can be reused. | Pipelines are designed to re-use functionality and utilities to achieve outcomes. |
| Publisher/Subscriber | Enable an application to announce events to multiple interested consumers | Information is provided to the system as the event happens and subscribers |

| Design Pattern type | Description | High level Implementation detail |
|---|---|---|
| | asynchronously, without coupling the senders to the receivers. | can process the events in almost real-time. |
| Queue-Based Load Levelling | Use a queue that acts as a buffer between a task and a service it invokes in order to smooth intermittent heavy loads that can cause the service to fail or the task to time out. | Queues are implemented to ensure delivery of messages and reliable performance where heavy loads are used. |
| Retry | Enable an application to handle transient failures when it tries to connect to a service or network resource, by transparently retrying a failed operation. | Retries are implemented where duplicated invocation does not affect the outcome of the functionality. |
| Schedule Agent Supervisor | Coordinate a set of distributed actions as a single operation. If any of the actions fail, try to handle the failures transparently, or else undo the work that was performed, so the entire operation succeeds or fails as a whole. | Application logic is implemented as a single agent to orchestrate independent processes and resolve any failures. |
| Throttling | Control the consumption of resources used by an instance of an application, an individual tenant, or an entire service. | Throttling would be incorporated to give resources priority access to the ADAP platform when available resources are low. |
| Valet Keys | Use a token that provides clients with restricted direct access to a specific resource, in order to offload data transfer from the application. | Keys of queues, functions etc. are stored in a secure vault and access is granted to the vault for use by outside applications. |

### 4.2.5 Development Standards

Azure has a multitude of coding languages supported on the platform depending on the service being used.  Languages used on the platform is selected based on the languages ability to deliver requirements, ease of implementation, language support and service used.

Python will be used as the primary language for development activities however other languages including C# and Javascript are being considered.

The tooling and extensions documented below are recommended.  Developers may choose to use their own preferred IDE however, it's the developer's responsibility to ensure that the IDE has all functionalities required for project delivery.

The coding standards defined in this document are mandatory and must be followed by developers on the ADAP.  This is to ensure that code development is consistent across the development team which will enhance efficiency in code development and code review, increase consistency in coding style and reduce maintenance effort.

#### 4.2.5.1 Python Development Standards

Python 3.6 is the version will be used across the platform.

##### 4.2.5.1.1 Recommended Tooling & Extensions

IDE: VSCode, Visual Studio
Extensions: pylint, Python, Azure Functions, GitLens, Azure Repos

### 4.2.5.1.2 Package Security

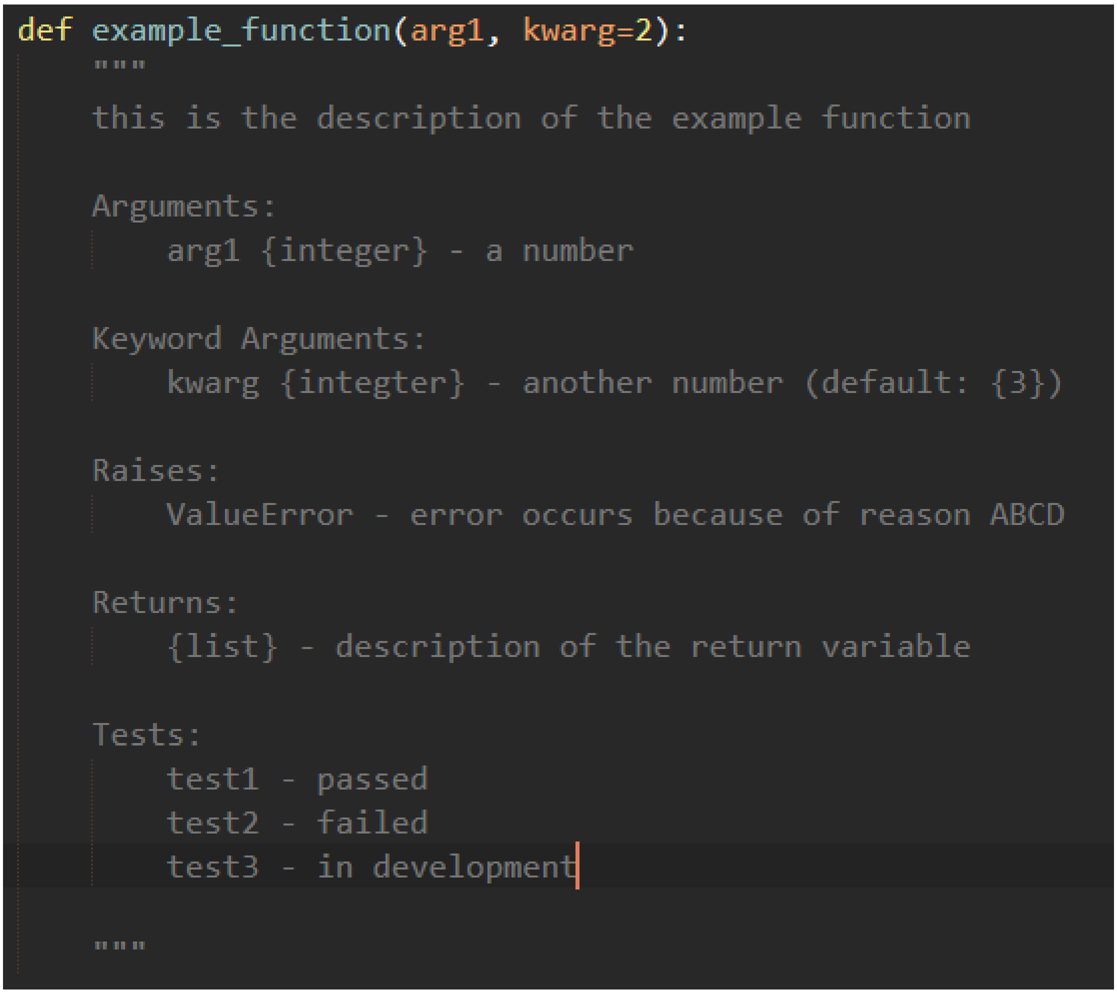
The security team have approved the usage of “safety” package for security vulnerability scan in Python. Developers will need to add any packages utilised in their code to a ‘requirements.txt’ file in the root folder that they deem is necessary to achieve the desired functionality provided that the packages passes the safety scan. This operation will be automated in the build pipeline check for all pipelines containing python packages.

In most cases, the scan will identify security vulnerabilities where the package version is out of date. In those cases, developers will have a chance to upgrade the packages and resolve any consequent bugs before recommitting their code.

In the case where the vulnerability is of another type (package too obscure to be identified, no version is secure, package is identified as unsafe, etc), the developer will work with the security team to identify a mitigating solution.

### 4.2.5.1.3 Coding Standard

- Style: use pycodestyle (formerly known as PEP8) https://pypi.org/project/pycodestyle/
  - line length is extended to 99 characters instead of 79
  - VS code has auto formatter that will help with code style. Packages autopep8, yapf, and black are are suitable.
- Package version: every root folder must contain a file named “requirements.txt” which include all packages used in this app with version number. Test packages should be grouped together at the end of the file. This file may be empty if no packages are needed.
- the default python logging package will be used for logging https://docs.python.org/3.6/library/logging.htm
- Docstrings:
  - Each file must contain file level doc string at the top of the file with a brief description of what the file is for.
  - Each operational function must contain a docstring with a short description of the function and inputs. Docstrings must also indicate if any tests are associated with the function, and if so, indicate the result from the test cases. Autodocstring can be used to simplify this process in VSCode. Example DocString Below:



```python
def example_function(arg1, kwarg=2):
    """
    this is the description of the example function

    Arguments:
        arg1 {integer} - a number

    Keyword Arguments:
        kwarg {integter} - another number (default: {3})

    Raises:
        ValueError - error occurs because of reason ABCD

    Returns:
        {list} - description of the return variable

    Tests:
        test1 - passed
        test2 - failed
        test3 - in development

    """
```

    - Tests will also include docstrings, but only a basic description of the test needs to be included
- Imports: all imports will be at the top of the file after the file-level docstring. Imports must be explicit (i.e. import * is not allowed)

#### 4.2.5.1.4 Function App Structure

The Lambda function code consists of at most three layers: the **main function**, the **business-level functions**, and the **util package functions** (*the utils are self-defined, not third-party packages*). Each layer should only call functions from the layer directly below it.

Main Function:

- This is controller for the function
- this is the last function defined at the bottom of the __init__.py in the root directory
- it should be high-level, concise and business readable and should not contain low-level technical details
- it should only make calls to the business-level functions and not any util package functions

Business-level Functions:

- Each business-level function should have a long, self-explanatory name, and sit in the __init__.py file above the event handler.
- They should be specific to the current function app, and should perform the bulk of the work.
- They should only call util package functions and not call other business-level functions.

Examples: get_config(), construct_patient_view(), send_result_to_frontend()

Util Package Functions:

- These are also referred to as helper functions in the industry
- The util functions should contain granular code at the lowest level

Examples: authenticate_to_sql(), parse_bytes_to_df()

#### 4.2.5.1.5 Logging

Logging will be required on all business level function calls during runtime. Util function do not require logging at runtime. This is to provide monitoring and traceability for defect investigation and resolution and performance monitoring. Each function with Function App should accept an environment variable named “verbosity” which will determine how much information is logged.

Verbosity Level:

- The verbosity level will be supplied to the function as an environment variable
- Non-prod environments should pass in the “debug” verbosity level
- Prod environments should pass in the “info” verbosity level

Required “info” logging:

- Business level function started (no parameter logging required)
- Business level function finished (no return value logging required)

Required “debug” logging:

- Business level function started with parameters provided
- Business level function finished with return values
- HTTP Request sent with payload
- HTTP response received with response

#### 4.2.5.1.6 Testing

The general testing framework, components to be tested and how test will be carried out will be detailed in section 7 TEST. This section will only detail coding standards related to testing within python. Pytest will be the testing package to be used for all python testing.

Unit tests:

- All unit tests must sit in “unit_test.py” in the root folder
- Every business-level function must have unit test and test all program paths and mock all HTTP requests
- There must be a pycodestyle check include in the unit test

Integration-tests:

- All integration tests must sit in “integration_test.py” in the root folder
- Every Azure Function must have integration tests that run at the __init__.py level with real requests.
- Test should include connectivity test to all nearest neighbours.

### 4.2.6 Naming conventions

All naming conventions will be lower case unless explicitly stated otherwise. All <descriptive contexts> may use camelCase for multiword names.

#### 4.2.6.1 Azure Service Names

AV has an existing standards and governance document which includes naming conventions and abbreviations for all resources and components on the Azure tenancy. Because he ADAP resides on AV’s existing Azure tenancy, all components on this platform will align it’s the abbreviation and naming conventions to said document where possible.  For any abbreviation not mentioned in that document, Azure's naming convention and abbreviations will be used

The following list contains the list of naming convention for services expected to be used in ADAP at the time of writing:

| Service | Naming Convention | Example |
|---|---|---|
| Azure Service Bus | sb-<subscription>-<airport location code>-<environment>-<project>-<descriptive context> | sb-npd-mel-dev-adap-ingestion |
| Azure Data Factory | df-<subscription>-<airport location code>-<environment>-<project>-<descriptive context> | df-npd-mel-dev-adap-ingestion |
| Azure API Management | apim-<subscription>-<airport location code>-<environment>-<project>-<descriptive context> | apim-npd-mel-dev-adap-internal |
| Azure Data Lake | dl-<subscription>-<airport location code>-<environment>-<project>-<descriptive context> | dl-npd-mel-dev-adap-archive |
| Azure function Apps | fa-<subscription>-<airport location code>-<environment>-<project>-<descriptive context> | fa-npd-mel-dev-adap-mdm |
| Azure SQL Database | sqldb-<subscription>-<airport location code>-<environment>-<project>-<descriptive context> | sqldb-npd-mel-dev-adap-dw |
| Azure Cosmos DB | cosdb-<subscription>-<airport location code>-<environment>-<project>-<descriptive context> | cosdb-npd-mel-dev-adap-main |
| Azure Databricks | adb-<subscription>-<airport location code>-<environment>-<project>-<descriptive context> | adb-npd-mel-dev-adap-mltransform |
| Azure Event Hub | eh-<subscription>-<airport location code>-<environment>-<project>-<descriptive context> | eh-npd-mel-dev-adap-ingestion |
| Azure Stream Analytics | sa-<subscription>-<airport location code>-<environment>-<project>-<descriptive context> | sa-npd-mel-dev-adap-avltransform |
| Azure Key Vault | kv-<subscription>-<airport location code>-<environment>-<project>-<descriptive context> | kv-npd-mel-dev-adap-main |
| Azure Machine Learning Service | ml-<subscription>-<airport location code>-<environment>-<project>-<descriptive context> | ml-npd-mel-dev-adap-predict |
| Azure Kubernetes Service | aks-<subscription>-<airport location code>-<environment>-<project>-<descriptive context> | aks-npd-mel-dev-adap-modelXX |
| Azure Application Gateway | ag-<subscription>-<airport location code>-<environment>-<project>-<descriptive context> | ag-npd-mel-dev-adap-lbfa |
| Azure Monitor | am-<subscription>-<airport location code>-<environment>-<project>-<descriptive context> | am-npd-mel-dev-adap-main |
| Azure Cache for Redis | redis-<subscription>-<airport location code>-<environment>-<project>-<descriptive context> | redis-npd-mel-dev-adap-spogcache |
| Azure Log Analytics | la-<subscription>-<airport location code>-<environment>-<project>-<descriptive context> | la-npd-mel-dev-adap-main |
| Azure Web App | wa-<subscription>-<airport location code>-<environment>-<project>-<descriptive context> | wa-npd-mel-dev-adap-spog |

### 4.2.6.2 Azure Key Vault

Key vaults will use the following naming convention.

| Secret Type | Naming Convention | Example |
|---|---|---|
| Key | key-<component using the key to encrypt>-<component using the key to decrypt>-<user group accessing the key>-<key type>-<key size> | key-fa-spog-service-rsa-2048 |

| Secret Type | Naming Convention | Example |
|---|---|---|
| | The user group should match a user group in AAD directly.  If the key is used for inter-service communication where no human is involved then the word “service” will be used | |
| Secrets | sec-<component sending the secret>-<component receiving the secret>-<account associated with the secret>-<user group accessing the key><br>The user group should match a user group in AAD directly.  If the secret is used for inter-service communication where no human is involved then the abbreviates “sp” or  “sas” will be used which represents service principal or shared access signature respectively. | sec-fa-sql-admin-sp |
| Certificate | cer-<component using the key to encrypt>-<component using the key to decrypt>-<user group accessing the key>-<key type>-<key size><br>The user group should match a user group in AAD directly.  If the key is used for inter-service communication where no human is involved then the word “service” will be used | cer-fa-spog-service-rsa-2048 |

#### 4.2.6.3 Azure Cosmos DB

Azure Cosmos DB will use the following naming convention.

| Entity Type | Naming Convention | Example |
|---|---|---|
| Database | db-<project name>-<”raw”/”curated”/”consumption”/”monitor”/”config”><br>The three keywords refer to the amount of processing the files sitting in each container will be.<br>Raw – “untransformed” file from the source.  File format can be changed but the content of the file should be the same as the source<br>Curated – files are cleaned using data quality and MDM controls.<br>Consumption – files are transformed, combined, or filtered to be used by a digital layer application.<br>Monitor – database used to store logs<br>Config – database used to store configuration files shared across the platform | db-adap-raw |
| Containers | Con-<descriptive context> | con-eventMessage<br>con-patientView<br>con-caseView |

Items within Cosmos DB are stored directly in their native JSON format.  The attribute “id” and the attribute selected for the partition key will uniquely identify an item within a container.  Items do not have names and thus does not require naming conventions.

#### 4.2.6.4 Function Apps

All functions within Function Apps will use the following naming convention irrespective of the capability layers they reside in. Because each Function App is specific and is not changing according to circumstance, the naming convention will be prescriptive.

| Function Grouping | Naming Convention |
|---|---|
| Data control checks | dq-checks |
| MDM control checks | mdm-checks |
| Data movement | data-movement |
| Utility | utility |
| Data processing and transformation | transformation |

#### 4.2.6.5 Azure Data Lake

Azure Data Lake is currently used as an archival mechanism for ADAP. For archival data, the container must be called “archive”. It is assumed that the same data lake will not be shared between ADAP and another project. The folder structure and naming convention to be used are as follows.

archive\<data source name>\<year>\<month>\<day>\<hour>\<minute>\<datetime>_<filename>.<file extension>

#### 4.2.6.6 Azure Service bus

Azure Service Bus will use the following naming convention.

| Entity Type | Naming Convention | Example |
|---|---|---|
| Queue | q<descriptive context> | qDurableFA1 |
| Topic | t<descriptive context> | tEventMessage |

#### 4.2.6.7 Azure Data Factory

Azure Data Factory will use the following naming convention.

| Entity Type | Naming Convention | Example |
|---|---|---|
| Linked service | ls_<environment>_<service name>_<descriptive context> | ls_dev_fa_dqActiveCheck |
| Dataset | ds_<environment>_<dataset type >_<descriptive context> | ds_dev_csv_Christ21Config |
| Pipeline lines | pl_<descriptive context> | pl_ingestBatchData |
| Activities | <descriptive context> | getConfigFile |
| Integration Runtime | ir_<country abbreviation>_<region name> | ir_au_east |
| Triggers | tr_<descriptive context>_<day of occurance>_<time of reoccurrence><br>day of occurrence and time of reoccurrence may be omitted if not applicable | tr_every_minute<br>tr_daily_8am<br>tr_weekly_monday_5pm<br>tr_monthly_2rd_2am |

#### 4.2.6.8 Azure SQL Database

Azure SQL Database will use the following naming convention.

| Entity Type | Naming Convention | Example |
|---|---|---|
| tables | tmp/stg/dim/fact/hub/link/sat_<descriptive context> | tmp_patientInfo<br>stg_patientInfo |
| Stored procedure | sp_<action verb>_<descriptive context><br>it's recommended that the descriptive context includes the type of sql resource being used (dim/fact/view/sp etc) | sp_update_dimPatient |
| Indexes | idx_<"c"/"u">_<table name>_<primary columns><br>"c"/"u" refers to clustered and unclustered respectively | idx_c_DimProduct_productCode |
| Sequences | sq_<descriptive context> | sq_mergeKey |
| Functions | fn_<descriptive context> | fn_alphaOnly |

### 4.2.6.9 Repositories

Repositories will use the following naming convention.  If a project does not have any capabilities, the word "main" will be used.

| Naming Convention | Example |
|---|---|
| <project name>-<capability name> | adap-setup |

### 4.2.6.10 Branches

Each repository will have 4 protected branch each representing an environment.  Each sub branch will be based off a protected branch and directly linked to a ticket on the DevOps board.  The sub branch would include the ticket number in the branch name.  The naming convention are as follows:

| Environment Name | Environment Abbreviation | Git Protected Branch Name | Sub-Branch Name Xx |
|---|---|---|---|
| DEVELOPMENT | DEV | dev | feature-XX |
| TEST | TST | tst | bugfix-XX |
| STAGING | STG | stg | bugfix-XX, release-XX |
| PRODUCTION | PRD | master | hotfix-XX |

For detailed explanation on branch definition, how the branches fit into the DevOps process and branch permissions, please refer to sections 5.9.6-5.9.8

### 4.2.6.11 Build pipelines

Build pipelines will use the following naming convention.

| Pipeline Type | Naming Convention | Example |
|---|---|---|
| Build pipelines | build-<capability name>-<service name>-<descriptive context> | build-ingestion-fa-diWriteToSQL |
| Automatic test pipelines | auto-<capability name>-<service name>-<descriptive context> | auto-ingestion-fa-diWriteToSQL |

### 4.2.6.12 Release pipelines

Release pipelines will use the following naming convention.

| Entity Type | Naming Convention | Example |
|---|---|---|
| Pipelines | release-<capability name>-<service name>-<descriptive context> | release-common |

| Entity Type | Naming Convention | Example |
|---|---|---|
| | <service name> and <descriptive context> are optional parameters. It may be necessary depending on how many release pipelines are configured for each capability layer | |
| Stages | <environment abbreviation>-<service abbreviation>-<descriptive context> | dev-fa-dqCheck |

### 4.2.7 Data quality and Master Data Management controls

The vision for ADAP is to be the central data servicing platform for all of AV. Data integrity and accuracy is the top priority for the platform. A series of data quality and MDM controls outlined below will be implemented to ensure that data landing in the platform is as accurate and complete as possible.

Anytime a control detects an anomaly, the result will be logged and the relevant business data owner will be notified of the issue.

**Data quality Controls**

| Control Name | Description |
|---|---|
| Completeness | Each dataset will have a pre-defined minimum number of attributes/columns that must exist on every ingestion. A check will verify if the minimum is met |
| Integrity | Each dataset will have a set of valid data types and data range for each attribute/columns. A check will verify if the type and ranges are satisfied. |
| Timeliness | Each dataset will have an expected time of arrival. A check will verify that the data has arrived within the time frame.<br>For real time data, there are a variety of checks that can be implemented. One example is listed below:<br>Number and type of packets per minute will be monitored as a proxy for ESTA CAD feed health. A sudden decrease in this count may indicate poor health in either the ESTA CAD source feed or the on-premise ESB |
| Consistency | For datasets that exists across multiple systems, data will be verify across the systems to ensure consistency. If a discrepancy exists, MDM controls will be applied to identify the source of truth and use that data in the platform. |

**MDM Controls**

Apart from data quality controls, Master Data Management is another key aspect of providing a robust data servicing platform. AV is currently undertaking a MDM project to identify the gaps between current state and target state. As part of the analysis, a series of MDM rules and controls will be identified. It is expected that ADAP will need to implement these controls natively until a decision is made on the final state MDM plan and tooling.

The current state analysis for MDM has identified data domains and business elements that have challenges in identifying the master and reference data source. While the scope of the ADAP is not to resolve MDM challenges with source systems, it can assist identifying discrepancies between the different sources.

A series of MDM rules will be built on the platform using Function Apps and will be incorporated into the data ingestion and data transformation stages based on rules identified and agreed upon with AV business stakeholders. Any discrepancies identified based on these agreed upon rules will be raised to AV stakeholders using Azure Monitor and emails.

### 4.2.8 Repository & Branch Structure

#### 4.2.8.1 Repository Structure Principles

Given ADAP will be an end to end platform spanning across all aspects of data analytics from ingestion to visualization, having the right repository structure is particular important.  The framework uses the following list of key consideration as the guiding principle for identifying the repository structure.

**Principle 1: Least privileged access**

The repository structure should logically separate different parts of the platform into isolated components so that developers both internal and external have access only to what they need and nothing more.  This is in line with industry best practice and helps to improve code security.  The ADAP is also presented with the unique challenge of incorporating third party providers as part of the development team in the future.  Aligning to this principle will ensure that third party developers should only receive access for the code base that's necessary for them to complete their tasks.  It will also help to protect AV's IP.

**Principle 2: Maintainability**

As the project will be use agile methodology with sprints, the mindset, structure and frequency of the SDLC is very different to that of a traditional waterfall project.  DevOps aim to automate as much of the delivery process as possible to improve efficiency and maintainability.  The repository structure should also support that framework and enable the code to be clearly structured and easily maintained.

**Principle 3: Expandable in the future**

The vision for ADAP is to become a data service layer for AV.  SPOG will become one of the downstream applications in the digital service layer.  AV is also expecting a variety of additional applications to be built on the top of ADAP as well as other use cases such as ad-hoc reporting and experimentations.

The repository structure must be able to cater for these expansions in the future.  It requires the flexibility to expand to additional cloud and on-premise components with minimal disruption to any existing development activity and minimal rework of any build & release pipelines.

#### 4.2.8.2 Repository Structure Options

Using the three key principles described above, three repository structure options were considered:

**Option 1: Single repository for the entire project**

This option can accommodate future expansions with the correct folder structure.  However, as the number of services and projects increase in the future, the folder structure can become overly complicated to navigate and maintain.  The depth of the folder structure will also have a directly impact on code development and code review complexity.  Furthermore, any internal developer requiring access to a specific part of the project will also receive full access to the entire code base.  This directly violates principle three.  While submodules (discussed in section 5.9.5) can be used to restricted internal access, this creates even higher maintenance overhead.

**Option 2: Separate repositories by capability layer**

This option promotes a logical separation of the project into different capability layers.  The exact definition of the layers are discussed in the following section.  This option will also utilize folder structures to help further segregate services within capabilities.  The maintenance overhead is reduced in that folder depth would remain small.  Because there is logical separation, components that would naturally be modified together will reside in the same repository (e.g. Service Bus and Function Apps called by the Service Bus).  Developers, internal & external, will also be receiving the correct level of access at the repository level.  There is the possibility of further restriction of access using submodules for third party developers, this is also discussed in section 5.9.5.

**Option 3: Separate repositories by service**

This option would separate each instance of a service into a new repository.  i.e. each Function App, containing multiple functions, would necessitate a new repository.  While this is extensible for future projects and provides granular user access controls, the maintenance overhead is prohibitive as the project expands given the number of repositories that will exist.

**Recommended Option**

Combining all the comments above, a summary is presented in the following analysis matrix.

| | Option 1 | **Option 2 (recommended)** | Option 3 |
|---|---|---|---|
| Principle 1 | ✓ | ✓ | Difficult to achieve |
| Principle 2 | ✓ | ✓ | ✓ |
| Principle 3 | Difficult to achieve | ✓ | ✓ |

Given option 2 provide the best fit to all the principles, it is the recommended option.  The following section will be discussed with the assumption that option 2 is used.

### 4.2.8.3 Description of capability layers

Option 2 requires the services on ADAP to be logically grouped together into capabilities.  The table below will detail the capability layers, their description and which services are expected to reside in each layer.

| Capability Layer Name | Description |
|---|---|
| Setup | This layer represents the infrastructure setup for all services used.  Access to this repository will be locked down and any change to this repository will require CR approval. |
| Common | This layer will include all data storage services and any service that will be shared across multiple repositories in the future.  In the case of ADAP, this capability will include data storage services only (Azure Data Lake, Azure SQL Database & Azure Cosmos Database) |
| Ingestion | This layer will include all services used for ingestion. |
| Transformation | This layer will include all services used for transformation.  Transformation performed within a storage service will not be included in this capability. |
| Machine Learning | This layer will include all services used for machine learning capability. |
| Data Access Management | This layer will include API management and any Function Apps used for the data servicing capability. |
| Configuration | This layer represents all configurations used by all aspects of the platform. |
| SPOG | This layer will include web apps used by SPOG |

For services that are used across multiple capabilities, the repository will only contain the instance relevant to the capability the service is used for.  For example, Function App is used both in ingestion and in transformation.  The Function Apps for ingestion and transformation will be two different resources within Azure.  Their code will also reside separately in the respective capability repositories.

#### 4.2.8.4 Repository Structure Detail

Repositories will be setup per capability as described in the option analysis.  The table below will detail out the capabilities, corresponding repository names and code content within each repository.

| Capability Layer Name | Repository Name | Repository Content |
|---|---|---|
| Setup | adap-setup | This repository contains all the infrastructure deployment ARM templates.  It is expected that this repository should not be updated often.<br>Security configurations for SQL database RBAC polices will also exist in this repository<br>Access to this repository will be strictly controlled. Changes to this repository will require change request approval process. |
| Common | adap-common | This repository will contain code for all the data storage services (SQL, Cosmos DB, Data Lake) and can be expanded to include any additional service that will be shared across multiple other repositories in the future.<br>SQL code will include all aspects (tables, views and stored procedures) except for security. |
| Ingestion | adap-ingestion | This repository will include code for services required for ingestion (Service Bus, Data Factory, Function App, Data Bricks).  In the future, it will also include Event Hub and Stream Analytics.<br>Only Function Apps & Data Bricks needed for ingestion will be included in this repository.  E.g. data quality controls, MDM, write data into storage. Function Apps used for transformation & data service into visualisation layers will sit in separate repositories. |
| Transformation | adap-transformation | This repository will include code for services required for data transformation (Function App & Data Bricks).  These include code logic that’s required to transform data into a usable format by one of the downstream applications.<br>Transformation performed by SQL or Cosmos DB using stored procedures will not be included in this repository. |
| Machine Learning | adap-ml | A separate repository is used specifically for code related to the ML capability on the platform (ML services, Kubernetes Services, Container Services). |
| Data Access Management | adap-data-access | This repository will contain code for API Management and any Function App used for data retrieval from ADAP into downstream applications. |
| Configuration | adap-configuration | This repository will contain all configurations used by all aspects of the platform. |
| SPOG | adap-spog | This is a designated repository for SPOG implementation. |

The following is configured in additional to the repository setup above:

- Every service within the repository will have their own folder.
- For Function Apps only, each Function App will have a sub folder in the main “Function App” service folder.
- If additional applications are built, they will be setup as separate repositories similar to SPOG setup.

#### 4.2.8.5 Sub-modules usage

In the case where AV wishes to further restrict code sharing to third parties from a capability down to a service level, the current repository structure can support that segregation by implementing submodules. Submodules is a standard Git functionality and allow a Git repository to exist as a subdirectory of another Git repository. For more details, please refer to https://git-scm.com/book/en/v2/Git-Tools-Submodules

When a third party requires access to a specific service only within the capability, which is stored as a folder in the git repository, that folder can be changed into a submodule. The third party will then be provided access to this submodule only. From the third party’s perspective, the submodule is the exact same as a repository. Their development process will remain the same. For developers already using the existing repository, some adjustments will need to be made to the commit process to incorporate submodules.

If a submodule is used, numerous configuration changes will need to be made to the DevOps components **for each submodule**. While this process is not difficult, it can be tedious.

1 A change request will need to be raised to convert a folder into a submodule.
2 The folder is converted into a new repository.
3 The folder within the existing repository is removed and replaced with the link to the new repository.
4 Git history and code accuracy are checked.
5 Existing developers will need to be trained on usable of git submodules. This is a one time activity.
6 Existing developers will need to pull the code to update their repository.
7 Build pipelines will need to be re-configured for converted folders. A complete reconstruction will usually not be required.
8 Release pipelines will usually not require modification as the build pipelines will construct the same artefacts stored in the same location
9 Build and release pipelines will need to be tested.

Once development is completed, it’s recommended that submodules remain as they are. This is done for two reasons:

- It’s likely that future third parties might also be engaged to work under the same service
- Converting the submodule back into the folder will require the inverse change process as stated above. This is again tedious.

Given these considerations, submodules is not recommended but is an option if code isolation is extremely important.

#### 4.2.8.6 Key considerations on the repository structure

The following key considerations have been noted regarding the repository structure noted above:

- **Code deployment of multiple languages in a single repository** is handled by creating separate folders in the repository for each function and corresponding language. Each folder will have its own Build script which will be required regardless of which repository option is chosen (single vs. multiple).

- **Build and release pipeline** across multiple repositories is handled through configuration files which contain the orchestration (sequence) of activities and tasks across repositories. These configuration files are stored in the 'Config' repo.
- **Maintainability of the multiple repositories** requires close oversight by Project and AV stakeholders to ensure that additional overlapping repositories are not created. The current repository structure has been developed with the principle that it covers the capabilities required from a standard data platform and therefore a large number of additional repositories is not anticipated. The repositories have also been structured such that they are modular in nature and have minimal (if any) overlap in terms of build and release. Where build and release activities are required across multiple repositories, this is handled by configuration files stored in the 'Config' repository (refer to Build and release pipeline above)
- **Re-use of existing ADAP repositories** is encouraged for all development on the ADAP. New repositories are only created on an exception basis and will require approval from AV Solution Architects and the KPMG Project Team.

#### 4.2.8.7 Environments & Branches

Gitflow is the industry standard branching strategy for the DevOps process. At a high level, it promotes continuous delivery using DEV as the main development branch and master as the production branch. Given AV's unique organizational function and its entailed responsibility, the branching strategy was adapted to include two additional branches (TST & STG) specifically used for carrying out different tests. Bugs raised will also have clear delineation between branches and environments.

For each repository, there will be four dedicated protected branch each corresponding to an environment: master, stg, tst and dev. There will also branches to manage feature development, bug fixes, hot fixes, sprint releases and pointing to the production environment. Sub branches are always tied to a ticket. Sub branch name will always contain the ticket number. Release branches will include the version number. The breakdown is outlined below

| Environment Name | Environment Abbreviation | Git Protected Branch Name | Sub-Branch Name Xx |
|---|---|---|---|
| DEVELOPMENT | DEV | dev | feature-XX |
| TEST | TST | tst | bugfix-XX |
| STAGING | STG | stg | bugfix-XX, release-XX |
| PRODUCTION | PRD | master | hotfix-XX |

Each environment and their corresponding branch is designed to serve the following purpose:

**Master Branch**: Production instance for any Azure service will be pointing to the "**master**" of the respective repository.
**Stg branch**: Staging instance for any Azure service will be pointing to the "**stg**" branch of the respective repository. The staging environment is intended to simulate a production deployment. The staging instance will be deleted when not in use to avoid the ongoing charges.
**Tst branch:** Testing instance for any Azure service will be pointing to the "**tst**" branch of the respective repository. The test environment will be used for UAT. The test instance may be deleted when not in use to avoid ongoing charges.
**Dev branch:** Dev instance for any Azure service will be pointing to the "**dev**" branch of the respective repository. The repository will be rebased as master to have multiple branches for continuous development of the features.

#### 4.2.8.8 Protected Branches

Every environment's dedicated branch is considered a protected branch. These branches are often linked directly to build and release pipelines that can change the physical Azure environment.

Therefore, commits cannot be made directly onto any protected branch. All code changes must be done via pull requests. This ensures that code are reviewed and pass through the appropriate testing before being potentially released onto any environment.

A protected branch is considered active if the environment is currently used. dev & master will always be active. tst and stg will only be active if UAT is occurring or release testing is occurring respectively.

The following branch policies will be applied the protected branches. Any sub branch making a pull requires into a protected branch must satisfy the following conditions.

| Branch Type | Branch Name | Require a minimum number of reviewers | Check for linked work items | Check for comment resolution | Automatically include code reviewers | Build Validation |
|---|---|---|---|---|---|---|
| Protected | master | 3 (DEV Admin) | Mandatory | Mandatory | Y | Y |
| Protected | stg | 2 (DEV Admin) | Mandatory | Mandatory | Y | Y |
| Protected | tst | 2 (DEV Admin) | Mandatory | Mandatory | Y | Y |
| Protected | dev | 1 (DEV Admin) | Mandatory | Mandatory | Y | Y |

#### 4.2.8.9 Sub Branches

These branches are tied to tickets and allow developers to make modifications to the code base to complete the requirement detailed in a ticket. As described earlier, XX in the branch name refer to a specific ticket number or version number in the case of release branches.

**Feature-XX**: Feature-XX branch will be created from DEV branch. Every feature will have its own branch. This will enable separation of code and versioning at every feature level.

**BugFix-XX**: BugFix-XX branch will be created from TST and STG branches. Any bug raised by the testers during feature testing and release testing will be addressed using this branch. Once fixed and tested successfully, the bugfix branch will be synced back to the respective protected environment branch to deploy the changes. These branches will also sync back to DEV to ensure the same error is not repeated in the future.

**Release-XX**: After the completion of UAT, a release branch will be created in the STG environment to maintain version control for the releases.

**Hotfix-XX**: If a bug is raised in the production environment and it requires urgent attention, Hotfix-XX (XX = relevant name or ticket number) branch will be used. Changes in hotfix branches will be synced back to all active protected branches.

**Sub branch deletion**: any sub branch merged into a protected branch after a successful pull request will be deleted.

**Archive-<Feature/BugFix>-XX**: Where a Feature or BugFix branch is created and is subsequently not required to be integrated into the codebase, the branch is renamed to include the term ‘Archive’ as a prefix to denote branch archival. As a principle, branches that are not integrated into the codebase are not deleted and instead archived.

#### 4.2.8.10 Branch Merge Approval

Dev Admin will have the rights to approve code merge onto protected branches (dev, tst, stg and master). Dev Admins are also expected to review the code before providing the approval. Dev Admin roles will change depending on which technology stack (data integration, database management, SPOG backend, SPOG frontend) are being deployed during each sprint. dev & tst code merge will require 1 approver and stg & master will required 2 approvers.

If the developers are experienced with the DevOps process and understand the requirements of the code review, it can also be appropriate to peer review the code for code merge onto dev & tst branches. This will reduce the bottle neck pressure placed on the Dev Admins to review all the code produced and increase delivery speed.

With the use of CICD and the new framework of DevOps, it is important to recognize that granting an approval for a code merge into a protected branch provides implicit consent for the code to be deployed into that environment. This is very different to the traditional waterfall model where the code merge and release are treated as two separate layers of control for a feature release. Approvers should review code with the intent that approval equates to production quality code.

## 4.3 Phase 3: Continuous Integration

Once a sprint starts, the development of each feature will use the following development and UAT process. Through the usage of Repos with Azure DevOps, all features will be integrated into the central protected branches continuously. The four high level stages of development are depicted in the diagram below.

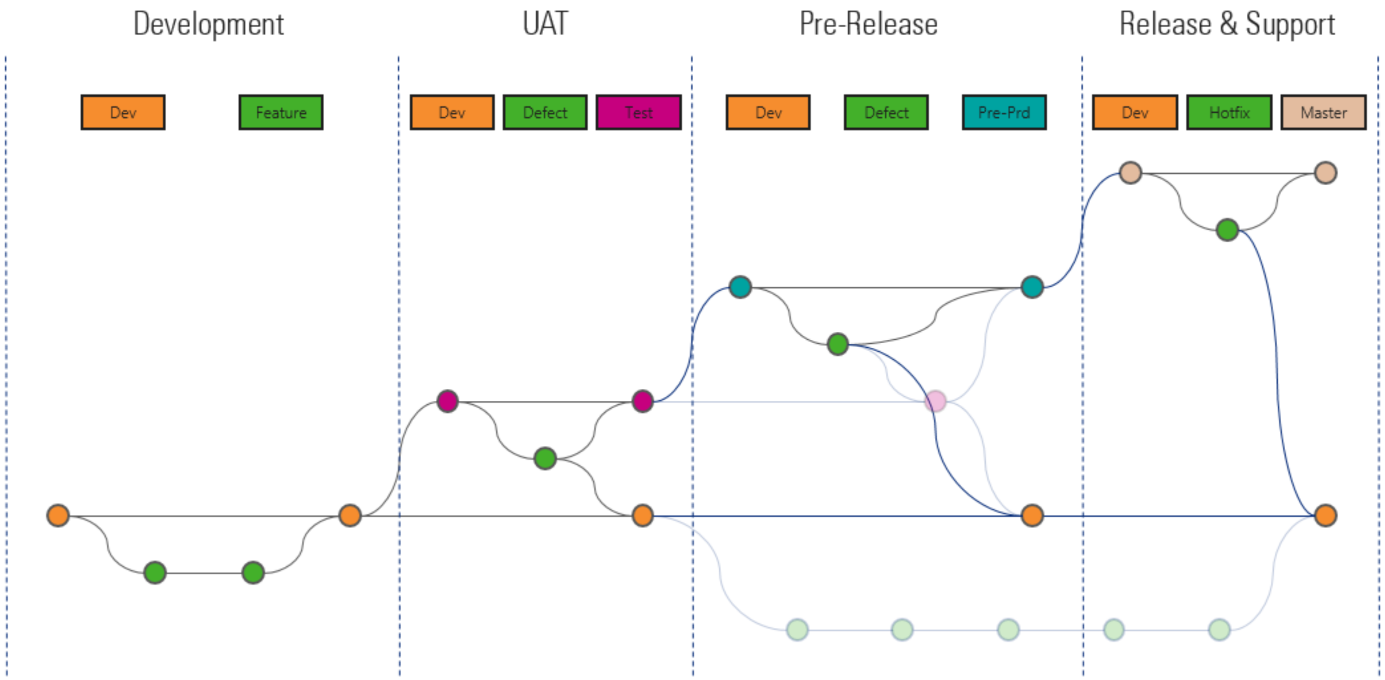
Once a sprint is planned, developers will be assigned tickets. Each developer will follow the key DevOps principles detailed earlier. Furthermore, each ticket will progress through the four development stages on the graph displayed on the right.

**Development**
Developers will implement the features required and carry out unit test(s)

**UAT**
UAT scenarios will be tested by business users. Any defects will be raised on Kanban and resolved.

**Pre-Deployment**
Once all UATs are passed, there will be a practice release into the STG environment. Any defects will be raised on Kanban and resolved.

**Release & Support**
Features will be released into the PRD environment with monitoring and support. Any hotfixes will be resolved through a defined resolution process.



### 4.3.1 Testing

The Testing section noted below is intended to be high-level only with separate documents created which detail information regarding the Testing Strategy and Test Plans (refer to Test Strategy and Test Plan documents)

Testing is a vital component of the DevOps process, and a strong focus on testing throughout all stages of the project will be instrumental to its success. Testing ensures that both functional and non-functional requirements are met and delivered solutions are of a high quality.

Test Driven Development will be used for the platform. The testing approach on this project employs a combination of unit testing, integration testing, regression testing and user acceptance testing throughout the development process. Unit, integration and regression tests will be fully automated, whereas user acceptance testing will be manual.

A mix of black-box and white-box testing will be implemented in order to evaluate both the functionality and the quality of the code respectively. White-box testing will focus on coverage testing and black-box testing will focus on meeting functional requirements.

Combinatorial testing may be used in the case where there this is a large number of combinations of variables to test on. Combinatorial testing will allow us to reduce the number of test cases whilst maintain a high level of fault detection. A requirements traceability matrix will allow us to produce relevant test cases that ensure the code meets requirements.

This section will detail out all the testing process and their resolution methods. There will also be references to specific branches in the git repository. Where these tests will be used in the development process will be clearly defined in the follow sections.

#### 4.3.1.1 Test Driven Development (TDD)

Test driven development is a test first method of development where requirements are turned into very specific test cases. Only after the tests are completed, then the code is written to ensure the tests pass. This ensures requirements are the main focus for every component.

##### 4.3.1.1.1 How will TDD be implemented

1. **Add a test**
   Write a test that defines a function or improvements of a function. The developer can accomplish this by covering the requirements and exception condition. Test will be written in the test framework that is appropriate to the software environment.
2. **Run all tests and check if the new test fails**
   This validates that the test is working correctly. The new test does not pass without requiring new code and it rules out the possibility that the new test is flawed and will always pass.
3. **Write the code**
   At this point, the only purpose of the written code is to pass the test. The programmer must not write code that is beyond the functionality that the test checks.
4. **Run tests**
   If all test cases now pass, the programmer can be confident that the new code meets the test requirements, and does not break or degrade any existing features. If the tests fail, the new code must be adjusted until all tests pass.
5. **Refactor code**
   The growing code base must be cleaned up regularly during test-driven development according to the development standards defined. By continually re-running the test cases throughout each refactoring phase, the developer can be confident that process is not altering any existing functionality.
6. **Repeat or stop**

Starting with another test, the cycle is then repeated to push forward the functionality if more functionality needs to be added. A trade-off needs to be made to ensure enough test coverage is achieved within a suitable time frame.

#### 4.3.1.1.2 Framework for TDD

The TDD framework requires 2 important factors when being used.

1. Consistent layout of usage to ensure the environment  is the same before and after tests
2. Reports generated from the tests in order to automate the testing process.

All tests should have the same layout which is shown below:

- Setup: Create the ideal conditions for the system needed to run the test.
- Execution: Trigger the target and capture all output, such as return values and output parameters.
- Validation: Ensure the results of the test are correct. These results may include explicit outputs captured during execution or state changes.
- Tear Down: Restore the overall test system to the pre-test state. This restoration permits another test to execute immediately after this one.

Different languages have different frameworks of testing, but there are ways of corroborating all tests. The framework chosen should produce a result for each component tested to ensure the tests can be automated. This generated result should also be uniform e.g. XML files produced from JUnit tests which gives a summary of tests passed or succeeded. These files can be produced by other language frameworks and a full summary of tests can be compiled to generate a report. The report is used by the automated framework to present a list of tests that failed and succeeded.

The following sections will detail how each type of testing will be used in the TDD framework.

### 4.3.1.2 Unit Testing

#### 4.3.1.2.1 Unit Testing Process

Unit tests are used to assess the functional correctness of a standalone piece of code before it is integrated into the development environment. They are the foundation of all testing in any development framework and will be a responsibility of all developers. On this project, unit tests will be developed by the developer working on a given feature, primarily completed during the initial development phase.

Unit tests are developed in the feature branch based on a specific function. This branch is based off the DEV environment, and will be merged back with DEV at the completion of development, once adequate testing has occurred and a pull request has been approved.

Unit tests must:

- Be automated
- Test the expected functionality of a function
- Provide full coverage on all test cases include edge and boundary cases

The tests must include docstring documentation within the function and be runnable through Build activity via the CICD pipelines. A list of language specific test frameworks will be provided in the coding standard for each language.  They can also be found at this link.

Only after the feature's development is complete and adequate unit testing has been conducted locally, should a pull request be submitted to merge the feature branch into DEV. Unit tests will be re-run alongside code review, and if defects are raised, further testing and development will be required.

After successfully merging with DEV, unit tests will be automatically conducted for every future build. Past this point, code integrity and alignment with the business requirements will be ensured by integration, regression and user acceptance testing.

### 4.3.1.3 Integration and Regression Testing

Integration tests group individual components together and tests their functionality as a group and regression tests aims at preventing the addition of a new feature or a bug fix introducing new bugs into the system. On this project, integration and regression tests will be initially created by a dedicated integration and regression tester. Having a single dedicated developer create these test cases removes that developer from the details and allows them to focus on the end-to-end macroscopic view of the system. This will driver better test cases development and higher test coverage. Follow-up modifications on these tests can be completed by developers when code is revisited and updated.

Any services that require integration with other components must have integration tests written during initial development. These tests will be reviewed and committed as part of the pull request. Integration and regression testing will be automated via the CICD pipelines upon new pull requests into protected branches.

One must first ensure that all unit tests have successfully passed in order to accurately evaluate the results of integration and regression testing. All relevant integration and regression tests must pass before a pull request is approved.

### 4.3.1.4 User Acceptance Testing

User acceptance testing brings the user into the testing framework. The goal is to ensure that the solution developed is suitable for the user. Selected users will create UAT scenarios and document them in Kanban before UAT commencement. The scenarios are aimed to test “real world scenarios” with expected results. Users are the ideal candidates to write the scenarios and test these cases because they will be the actual users of the system.

UAT will be the only form of manual testing in the testing process. Once all defects have been resolved and all UAT have been successfully passed, a pull request can be issued to STG.

#### 4.3.1.4.1 Creation of UAT scenarios

UAT scenarios will be written in parallel to development and tested by the relevant user groups within AV. Scenarios should be based on requirements as defined in the user stories on Kanban. A UAT plan will be created in order to outline the strategy used to verify and ensure features meet the business requirements. This plan will cover exit criteria, testing scenarios and test case approach, alongside a rundown for each test case that includes:

- Functionality
- The relevant user groups
- A step by step guide on how to use the platform to achieve this functionality.

All test plans will be documented in Azure Test Plans and Kanban where appropriate.

These UAT scenarios will be provided to the relevant user groups in which they will evaluate the correctness and quality of the solution using Azure Test Plans. Azure Test Plans provides the user an easy to use framework to evaluate whether the solution meets the requirements and if it doesn’t, at exactly what point did bug occur that prevented the test from passing. The test will also have the opportunity to provide feedback directly in Test Plan. These feedback are linked to the test plan ticket on Kanban automatically. This provides an immediate feedback mechanism with high visibility to the entire team allowing the developer to quickly address the bug.

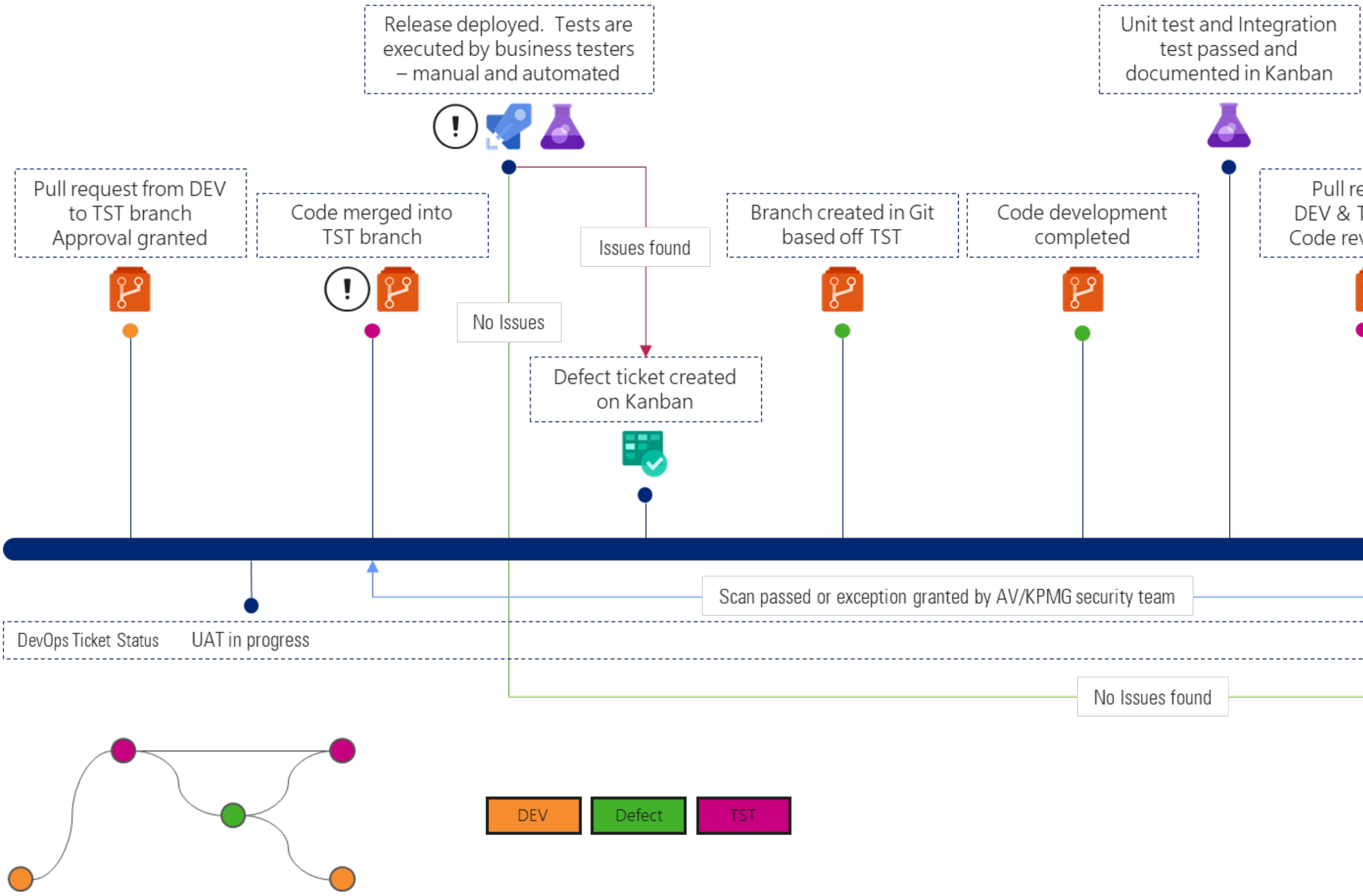
#### 4.3.1.5 User Acceptance Test Process

1. Users will be writing UAT scenarios in parallel to code development.  These must be completed before UAT can commence.
2. Once enough features have been developed and a UAT is scheduled, a pull request will be made to merge DEV into TST branch
3. Code review, automated testing and defect resolution will be completed.
4. After code is merge into TST branch, CICD pipelines will automatically build and deploy the changes to the test environment.  UAT will commence.
5. When an issue is found, a defect ticket will be created on Kanban
6. A branch will be created in git based off TST
7. Step 5 – 9 from the development process repeats.  Code is merged into both DEV and TST environment for every defect resolved
8. Step 12-15 repeats until all UAT test scenarios are passed

Code needs to be merged back into the DEV environment to ensure that developers are always working with the latest code.  This will ensure that any defects raised during the UAT process is not repeated in the future.

Ambulance Victoria

## UAT Process Diagram



Pull request from DEV to TST branch Approval granted

Code merged into TST branch

Release deployed. Tests are executed by business testers – manual and automated

No Issues

Issues found

Defect ticket created on Kanban

Branch created in Git based off TST

Code development completed

Unit test and Integration test passed and documented in Kanban

Pull re
DEV & T
Code rev

Scan passed or exception granted by AV/KPMG security team

DevOps Ticket Status UAT in progress

No Issues found

DEV

Defect

TST

### 4.3.1.6 Non-Function Requirement Testing

While all the tests described previously were used to test functional requirements, non-functional requirements must also be monitored and tested during the development process. NFR's that are to be tested (including accepted thresholds) as part of each User Story are documented in the design document and reviewed by Project stakeholders.

#### 4.3.1.6.1 What is NFR?

*A non-functional requirement (NFR) is a requirement that specifies criteria that can be used to judge the operation of a system, rather than specific behaviours. Broadly, functional requirements define what a system is supposed to do and non-functional requirements define how a system is supposed to be.*

#### 4.3.1.6.2 How will it be tested (Implementation)

NFR testing on the ADAP and SPOG is based on the key technical development areas on the platform. Development on ADAP and SPOG can be broadly divided into four key components (Platform; Ingestion; Machine Learning and SPOG). The table below provides a view of the types of NFR testing performed on these components

| Type of testing | Description | Platform | Ingestion | ML | SPOG |
|---|---|---|---|---|---|
| Baseline | Validation of the documents and specifications on which test cases are designed. | | ✓ | ✓ | ✓ |
| Compliance | Determine whether a process, product, or service complies with the requirements of a specification, technical standard, contract, or regulation. | ✓ | ✓ | ✓ | ✓ |
| Documentation | Black box testing that ensures that documentation about how to use the system matches with what the system does. | ✓ | ✓ | ✓ | ✓ |
| Endurance | Testing a system with a typical production load, over a continuous availability period, to validate system behaviour under production use. | ✓ | ✓ | ✓ | ✓ |
| Load | Process of putting demand on a system and measuring its response. | ✓ | ✓ | ✓ | ✓ |
| Localization | Product is checked to assure that it behaves according to the local culture or settings. | | | | ✓ |
| Performance | Determine how a system performs in terms of responsiveness and stability under a particular workload. | ✓ | ✓ | ✓ | ✓ |
| Recovery | The activity of testing how well an application is able to recover from crashes, hardware failures and other similar problems. | ✓ | ✓ | ✓ | ✓ |
| Resilience | Ensuring that applications will perform well in real-life or chaotic conditions. | ✓ | ✓ | ✓ | ✓ |
| Security | Process intended to reveal flaws in the security mechanisms of an information system that protect | ✓ | ✓ | ✓ | ✓ |

| Type of testing | Description | Platform | Ingestion | ML | SPOG |
|---|---|---|---|---|---|
| | data and maintain functionality as intended. | | | | |
| Scalability | Measure its capability to scale up or scale out in terms of any of its non-functional capability. | ✓ | ✓ | ✓ | ✓ |
| Stress | Deliberately intense or thorough testing used to determine the stability of a given system or entity. | ✓ | ✓ | ✓ | ✓ |
| Usability | The practice of testing how easy a design is to use on a group of representative users. | ✓ | | | ✓ |
| Volume | Analyse the system performance by increasing the volume of data in the system. | ✓ | ✓ | ✓ | ✓ |

#### 4.3.1.7 When it will be tested

Upon deployment of the application into the test (TST) environment, the environment would be exposed to users and developers to conduct a series of non-functional tests before the environment would be ready to be released into production.

Scenarios should be created before the testing commences to ensure specifications are met according to the goal of the test. NFR is a combination of manual and automated testing depending on the component deployed and will require both AV and KPMG resources to conduct this testing.

#### 4.3.1.8 How it will be tested

The following table defines in which environment each NFR test will be performed and whether it’s a manual (M) or automated (A) test.

| Type of testing | DEV | TST | STG |
|---|---|---|---|
| Baseline | ✓ (M) | ✓(M) | |
| Compliance | | | ✓(M) |
| Documentation | | ✓(M) | |
| Endurance | | ✓(A) | |
| Load | | ✓(A) | |
| Localization | | ✓(M) | |
| Performance | | ✓(A) | |
| Recovery | | ✓(A) | ✓(A) |
| Resilience | | | ✓(A) |
| Security | | ✓(A) | |
| Scalability | | ✓(A) | |
| Stress | | ✓(A) | |
| Usability | | ✓(M) | |
| Volume | | ✓(A) | |

#### 4.3.1.9 Defect Management Process

For the purpose of this document, defect and bug are used interchangeably.

All defects found by the testing processes discussed in the earlier sections above will follow the same defect management process outlined below.  This simplifies and centralizes the resolution process and provides higher transparency and auditability.

All defect found during feature development will only be visible to the developer themselves and hence are resolved by the developer locally. Any defects found in later stages of the development process (i.e. after the initial pull request to DEV) will be logged and managed through the incident escalation process as outlined in the support document. This general incident escalation process is outlined below:

1. When a bug is found or reported, a bug ticket is created.
2. If this bug is found during the development process, the demand manager will prioritize this bug.
3. If this bug is found in the production environment, it will be prioritized by service desk agent referencing The Project Icon Support Design Document as a guideline.
4. The defect might require additional triage depending on the type of ticket. E.g. security bugs will require security review.
5. The defect ticket will then have an action taken dependent on its selected priority level which is specified The Project Icon Support Design Document. The available actions are: known defects, ignore, bugfix, hotfix

After defects have completed the incident escalation process, they will be processed and resolved in the following manner depending on the action type:

**Known defects:**
A known defects registry will be maintained. Any new defects identified that is not addressed in the current sprint must be added to this registry. This will include all known defects regardless of if they are intended to be addressed in the future sprints. Once a defect is resolved, the resolution method summary and the ticket number must be maintained in this registry.

**Ignore:**
This action type is largely intended for vulnerability scans. Because the scan is automatic and out of context of external controls, it might identify security issues that are being addressed by another control (e.g. network security protocol). Any defects falling under this category must be security reviewed and approved.

For defects raised as part of the testing process, join approval from two Dev Admins must be obtained in order to classify a defect into this action type.

All defects under this category must also be added to the known defects registry along with the resolution justification and ticket number.

**Bugfix:**
This is for defects that are addressed directly as part of the sprint activity. These can apply for DEV, TST and STG branches. The following work flow will be used to address the bug:

1. Ticket logged on Kanban
2. A sub branch, based off the branch correlated with the environment containing the bug, is created and associated with the ticket
3. Development occurs to resolve the bug
4. A pull request is made to the protected branch
5. Code review and automated testing occurs. UAT will also be carried out if appropriated
6. If the bug is addressed, ticket will be closed and protected branch will be updated
7. If the bug is not addressed, step 4 and 5 are repeated
8. If the resolution of this bug creates a different bug, then step 6 is used for this bug and the new bug will follow the entire process from step 1 again.

**Hotfix**:
This is specifically for defects raised in the production environment. For the classification of when a defect will be classified as a hotfix VS a rollback on the entire production environment to a previous release, please refer to the support documentation. The following work flow will be used to address the hotfix bug:

1. A hot fix ticket will be created on Kanban
2. A branch will be created in git based off Master
3. Test cases and defect resolution will occur in parallel
4. After development is complete, all test cases will be executed. No UAT will be performed.
5. Any failure of tests will be addressed directly in the hotfix branch with comments in the ticket.
6. Once all tests are passed, a pull request will be made to master and DEV branches.
7. CICD pipelines will automatically deploy the new solution to production environment.

Hotfixes are typically deemed critical enough that it cannot go through the regular defect management process. These fixes are typically performed on the “hot” environment (i.e. Production) and do not go through the regular testing cycle. Changes that are deemed to be hotfixes require Product Owner approval prior to the change to the Production environment.

#### 4.3.1.10 Quality Metrics

A range of metrics will be used in order to ensure both the functionality of the code and quality of the code structure. Maintaining the quality of the code is important in order to more easily maintain code, fix bugs, add new features, and guarantee the efficiency and robustness of the solution.

Azure DevOps has a number of metrics relating to both code quality and functionality that can be visualised and analysed in order to maintain code quality throughout the development and testing process. We will be use the following Azure DevOps metrics to assess these criteria in this project.

- Total tests
- Number of test runs completed
- Number of test passed/failed/others
- Test pass percentage
- Run duration
- Number of tests not reported
- Number of unique failing tests in the last 14 days
- Number of new defects
- Changes in the above metrics from the last run

The above metrics specialise in evaluating the functional quality of the code. The use of these metrics along with code reviews and use of the requirements traceability matrix with UAT will ensure that the solutions being developed achieve both the function and non-functional requirements. Functional metrics will be monitored by developers writing and running testing. Non-functional metrics will be assessed primarily in UAT and coverage metrics collected from user behaviour on the platform upon different releases.

#### 4.3.1.11 Feature Development Process

Once a sprint starts, the development of each feature will use the following development and UAT process. Through the usage of Repos with Azure DevOps, all features will be integrated into the central protected branches continuously.

Below is a list of steps which will be followed in order to develop features and will happen in parallel for multiple features. A feature prioritisation activity will be performed in order to tag dependencies between features and sequence the feature development in order to have an efficient continuous integration exercise.

1. Feature ticket created on Kanban
2. Branch created in git based off DEV
3. Automated test cases are written
4. Test plan for UAT tests documented in Kanban
5. Once development completes, a pull request is submitted to merge the feature branch into DEV

6. Code review occurs, build activity with Azure DevOps will run all automated tests (unit test, integration test and regression test) and vulnerability scans are carried out
7. All defects raised from the previous step will pass through a review process.
   - For any redundant or known defects (according to a known issues registry), the defect may be waived by a senior team leader
   - All other defects will be logged into Kanban as tickets and prioritised to be resolved.
8. A pull request may be rejected based on the review results.
9. Once all defects are resolved or waived, the pull request will be completed and the code is merged into the DEV branch

### 4.3.2 Build agents

Build agents are used to run any build and release pipelines within Azure DevOps to deploy changes into the ADAP environments (DEV, TST, STG and PRD).  Build agents can be grouped into build agent pool for resource management purposes.  Azure DevOps provides 1 build agent pool containing 1 default hosted build agent free of charge.  The agent has the capacity to process two builds in parallel for free up to 1800 minutes per month on the free tier.  Additional parallel processing capacity or additional processing time can be purchased.

At the time of writing, the default build agent pool should be sufficient to accommodate all build requirements within the project.  As the project grows and more builds are more regularly required, build agent pool and build agents can be purchased.

Because build agents is an Azure DevOps admin level setting, only the DevOps service admin should have the ability to modify them.  Any changes to the build agent should be raised as a change request following AV's CAB process.

### 4.3.3 Build activities

Builds will be used in two different ways on this platform – automated testing & artefact building. Branch policy will be setup to require all sub-branches to complete build during a pull request into a protected branch.  All automated testing (unit, integration, regression) will be completed via a build stage during the build activity for pull requests.  Pipelines will also be setup to build on all protected pipelines on after every code merge to generate artefacts for deployment.

**Build Pipelines**
Build pipelines will be built using YAML and version controlled through GIT.  Because build pipelines control how artefacts are generated for the release process, only developers with the "DevOps Builder" role will have the rights to modify build pipelines.  Pipelines will utilise environment variables to reuse the pipeline across multiple environments.

**Artefacts**
During the build process, protected branches will usually generate artefacts which is used for environment deployment later.  Sub-branches usually do not generate artefacts.  Artefacts will have a retention period of 30 days and are managed automatically in Azure DevOps.

**Notifications**
Build pipeline failures will automatically trigger an email notification to the developer who initiated the build activity through the pull request.  It's the responsibility of that developer to resolve the bug causing the build failure.  This automatic notification enables fast feedback and facilitates fast fixes.

**Monitoring**
Azure DevOps will automatically collect build metrics and generate pre-defined reports.  These are available within DevOps.  As these are development metrics, it's appropriate for them to be displayed in DevOps only.

### 4.3.4 Build failure pipelines

Each job in every release pipeline will have, at a minimum, one job for deployment and one job for 'Build Failure' where a previous job has failed. The 'Build Failure' job will have an execution condition for 'Run this job' as 'Only when a previous job has failed'. The purpose of this job is to reverse any steps to restore the environment to the target state. An example could be to drop a table in the database or to restore a Function App to the previous version. Build failure pipelines are specific to each repository and the design and deployment of these jobs should consider the feature or functionality being deployed and the potential failures that can occur during the build process .

## 4.4 Phase 4: Deploy

### 4.4.1 Release Definition

While feature development is sprint based and driven by tasks, releases correlate to user stories. Therefore, releases might not occur after every sprint. The demand manager will work with the product owner and development team to define the release schedule based on user story prioritization, development difficulty and external factors (e.g. infrastructure maintenance schedule).

### 4.4.2 Release Approval

Release Approvers will have the rights to approve pipelines to deploy code changes into the physical environments. TST & STG release will require at least 3 approvers while PRD will require sign off from all approvers. If an approver is sick or out of office, it's the duty of the approver to find a replacement and notify KPMG to make the appropriate changes. Note a regular change request will have 5 days lead time after CAB approval. CABs are only held once a week on Tuesdays.

Release Approvers should always include a change manager, a product owner for the use case(s) being deployed, an AV solution architect, and an AV network & securities expert. The exact individuals will be agreed to between KPMG & AV.

Release approvers are not responsible for checking the functional requirement, non-functional requirement or the quality of the code. That has already been completed by the branch merge approval. The release approvers should ensure that the business is ready for the deployment – e.g. UAT scenarios has been written prior to deployment to UAT environment; or change communication have been sent to the business for deployment to PRD environment.

### 4.4.3 Release Pipelines

Release pipelines will be configured to deploy on all code changes for TST, STG and PRD environments. While developers DevOps Builder roles will also be able to modify release pipelines, modifications of release approvers will require Project Admin approval.

Release pipelines will usually automatically be triggered when a new designated artefact is available for TST and STG environment. Release will be manual for PRD environments.

Similar to build pipelines, a release pipeline failure will trigger an email notification to the initiator and release pipeline metrics are also stored in Azure DevOps only.

DEV environment does not have build or release pipelines configured. Developers can make code logic modifications directly into the environment.

### 4.4.4 Release Process

Once enough user story(s) have been developed and passes all the test cases, UAT and vulnerability scan, the following process will be used to release the new changes into the production environment. The release will be completed in two stages – staging release and production release.

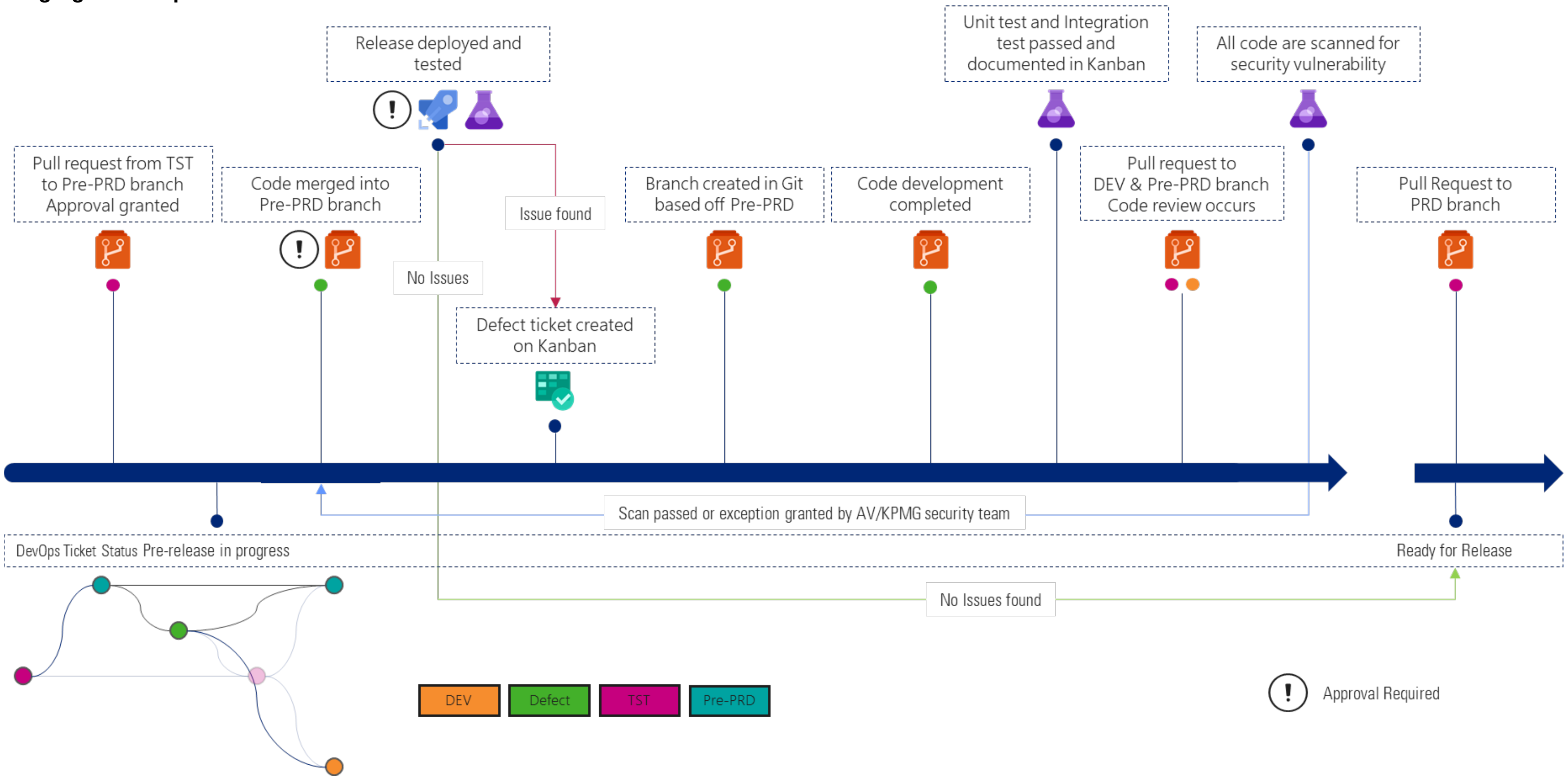
Staging release is specifically used to test the release deployment. Before the release, it will be configured as an exact copy of the production environment. After every staging release has completed, the previous version will also be deployed into the same environment. This is to ensure that rollbacks will always be deployable in the production environment if necessary.

#### 4.4.4.1 Staging release

Steps to complete the release post approval for release into the STG environment:

1 A pull request is made from the TST branch into the STG branch for all code relevant to the current release. Release approvers receive notification for code review.
2 Release approver confirm the code to be merged is correct and approves the pull request.
3 Code is merged into STG branch with the appropriate version number tagged.
4 CICD pipelines initiate automatic deployment of changes and send release approval requests to all approvers.
5 The designated number of approvals are received.
6 CICD pipeline deploy the changes to the Staging environment.
7 If an issue is identified, it will be raised on Kanban and evaluated through the Defect Management Process as detailed in section 5.5.
8 If at this point, it is identified that the issue was not identified or mitigated by the controls already in place, the controls (e.g coding standard, security scan) will be updated.
9 Once the STG branch is ready for merge into the Master branch, a pull request will be made from STG to DEV. This is done to ensure that any defects resolved in the STG environment are also applied to the development environment.
**10** Once code review is complete, the code will be merged into DEV branch.

## Staging release proces



Pull request from TST to Pre-PRD branch Approval granted

Code merged into Pre-PRD branch

Release deployed and tested

No Issues

Issue found

Defect ticket created on Kanban

Branch created in Git based off Pre-PRD

Code development completed

Unit test and Integration test passed and documented in Kanban

Pull request to DEV & Pre-PRD branch Code review occurs

All code are scanned for security vulnerability

Pull Request to PRD branch

Scan passed or exception granted by AV/KPMG security team

DevOps Ticket Status Pre-release in progress

Ready for Release

No Issues found

DEV

Defect

TST

Pre-PRD

Approval Required

### 4.4.5 Production Release

Production release will not always occur after every sprint. The release scheduled will be determined by KPMG in consult with AV. When production release is scheduled, the following process is used to release changes:

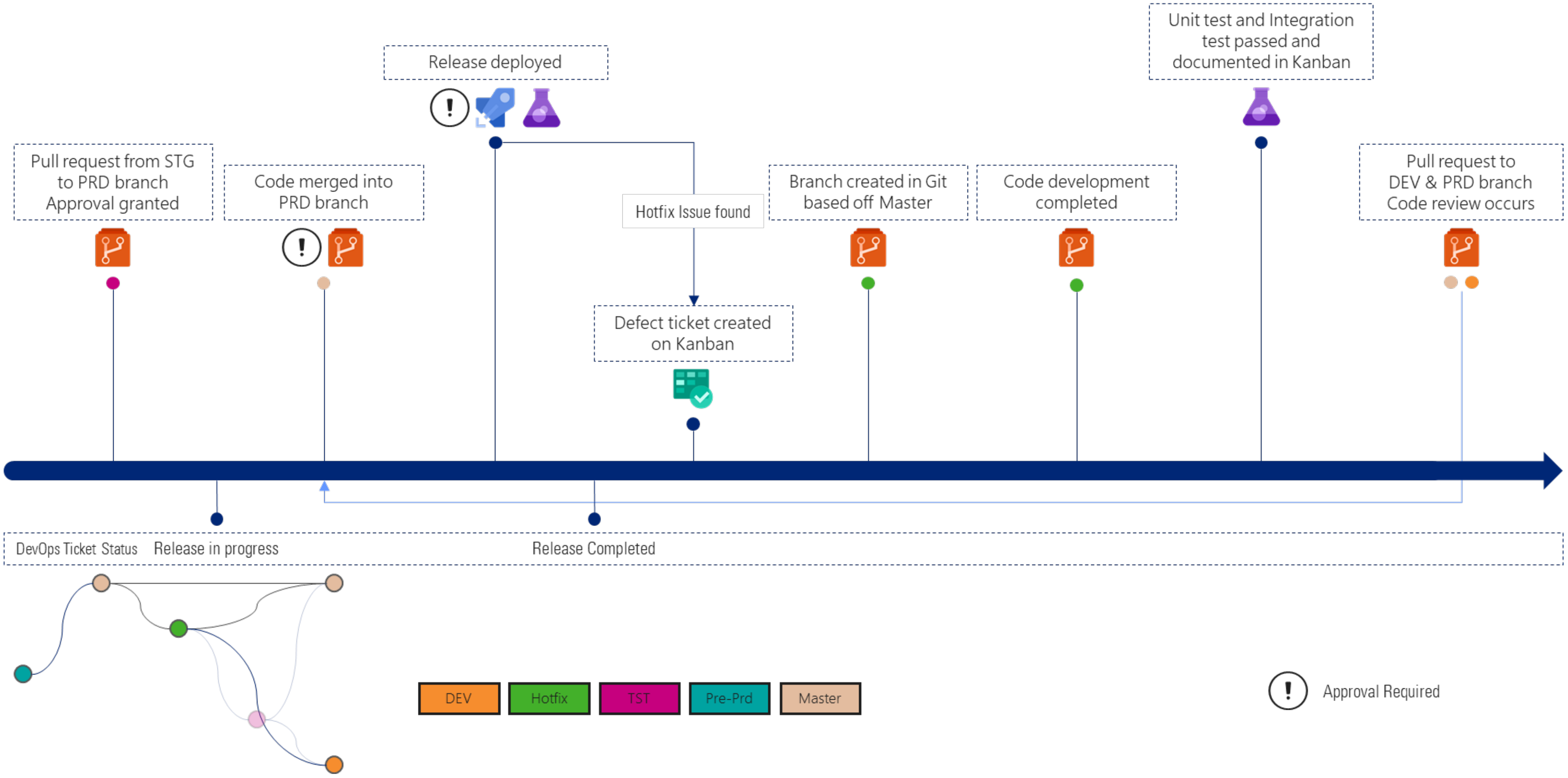
11 A pull request is made from the STG branch into Master branch
12 Code approver confirm the code to be merged is correct and approves the pull request.
13 Code is merged into master branch with the appropriate version number tagged.
14 CICD pipelines initiate automatic deployment of changes and send release approval requests to all approvers.
15 All approvers approves deployment request
16 CICD pipeline deploy the changes to the Production environment.
17 Service Health will be check after deployment

All code used for each release will be tagged with a version number. Because Azure DevOps has built in activities for Azure services deployments, database deployments will function the same way as any other services. Rollbacks can be easily achieved with the manual deployment of a previous version of the code.

### 4.4.6 Rollback & Hotfix Process

For certain defects where the size is large and priority is critical or where incorrect code has been deployed to the Production environment, it may be determined by the Product Owner to rollback the code to a previous iteration. The determination of whether the Production environment requires a rollback due to a defect is listed in the ‘Support Design and Service Management Plan’ document. Each stable Release to the Production environment is tagged with a version number in the STG master branch which can be used for rollback purposes.

## Release & Hotfix Process



Pull request from STG to PRD branch Approval granted

Code merged into PRD branch

Release deployed

Hotfix Issue found

Defect ticket created on Kanban

Branch created in Git based off Master

Code development completed

Unit test and Integration test passed and documented in Kanban

Pull request to DEV & PRD branch Code review occurs

DevOps Ticket Status Release in progress Release Completed

### 4.4.7 Change Management

Every release should be accompanied with the appropriate change management process to ensure that the affected parties are notified of the change. All releases on the ADAP will follow the existing AV change management process.

## 4.5 Phase 5: Operate

After all releases are completed, Azure Monitor will be used to continuously monitor the overall health and security of the services. This enables a constant feedback loop so that the platform can be continuously evaluated and improved. Monitoring will be enabled on all services for all metrics and logs. Relevant logs will be streamed into Azure EventHubs for AV on-premise consumption.

Some key metrics that will be monitored are:

- Service health
- Service status
- Login requests for all services
- Message process time
- Database query time
- Database index & table fragmentation statistics
- Function App utilization statistics
- Web App utilization statistics

Where appropriate and necessary, logs from multiple sources maybe aggregated or correlated together to create additional meaning monitoring criteria.

**Alerts & Notification**
To ensure that all issues are triaged quickly and reduce the impact of potential major issues, alerts and notifications will be setup to advise the support team when potential issues will need to be addressed. Alert criteria and threshold will be agreed with between AV and KPMG. Some notification criteria might be:

- Consistent long HTTP request time
- Consistent long data query query time
- Suspicious login activity
- Change in service status
- Change in service health
- Consistent high Function App utilization
- Consistent low Function App utilization

The exact alerts for each service will depend on the nature of the service and how it is used in the platform. They will be designed in consultation with AV during sprint planning.

Each agreed notification will have a response plan pre-defined in the support document. For detailed support operation process, please refer to the support document (link to be added when document is available).

### 4.5.1 Regulatory Compliance

Given the platform will be hosting PII data in aggregation, the platform is classified as “protected”. A comprehensive security and regulatory analysis was conducted as part of the solution architecture design process for the controls required (technical & business process) to secure the data and provision the right level of access.

On initial deployment, all controls and deployment methodologies will be reviewed by a security team member and signed off by both KPMG and AV security teams. Because users are able to access the databases in DEV and TST environment, all PII data will be anonymized before landing the in platform. The develop team will work with the security team to ensure that this is completed prior to landing any data in DEV or TST environment.

To ensure continued compliance, the following framework will be used:

**STG & PRD environment**

Every release into STG or PRD environment must pass network and security team review. The network and security team may recommend penetration testing or any other tests as part of their review process based on change being implemented or time since last test. The network & security team will be responsible to ensure that ADAP is and will be compliant to all regulatory requirement.

**DEV & TST environment**

For releases to the DEV & TST environment, all infrastructure deployments must also pass network and security team review because deployment affects how data is stored or how components communicate with each other.

Code deployments, in most cases, will not affect the security posture of the platform because code deployments usually affect the logic within service components. The joint approval from 2 Dev Team Leader may waive the requirement for a security team review for code deployments. This will need to be documented on Kanban.

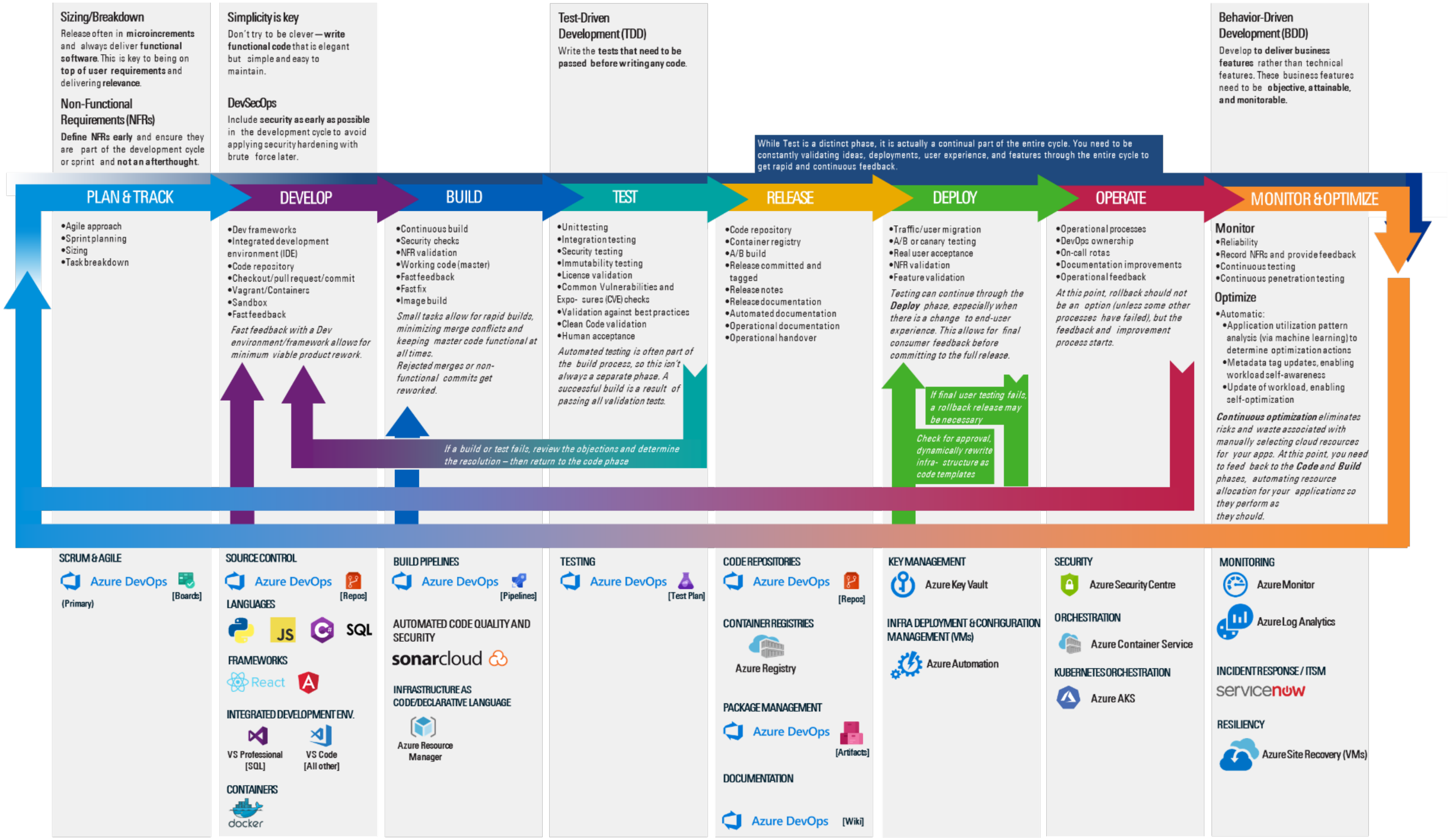
An example where security team review should be conducted would be ingestion of new data sets onto the platform. While all changes are code deployments, the new data must be anonymized before it can enter DEV & TST environments. These affect the security control required for the data and thus requires security team review.

## 4.6 Phase 6: Continuous Feedback

The continuous feedback process is a natural extension to the monitoring processes implemented in Phase 5: Operate. In addition to the technical feedback provided through monitoring mechanisms, a retrospective is held at the end of each Sprint for the team to identify improvement and automation opportunities. The metrics identified in the previous phases will be analysed to identify improvement opportunities.

# 5. Appendices

## 5.1 Appendix A – Detailed DevOps process and tooling



**Sizing/Breakdown**

Release often in **microincrements** and always deliver **functional software**. This is key to being on **top of user requirements** and delivering **relevance**.

**Non-Functional Requirements (NFRs)**

**Define NFRs early** and ensure they are part of the development cycle or sprint and **not an afterthought**.

**Simplicity is key**

Don't try to be clever — **write functional code** that is elegant but simple and easy to maintain.

**DevSecOps**

Include **security as early as possible** in the development cycle to avoid applying security hardening with brute force later.

**Test-Driven Development (TDD)**

Write the **tests that need to be passed before writing any code**.

While Test is a distinct phase, it is actually a continual part of the entire cycle. You need to be constantly validating ideas, deployments, user experience, and features through the entire cycle to get rapid and continuous feedback.

**Behavior-Driven Development (BDD)**

Develop **to deliver business features** rather than technical features. These business features need to be **objective, attainable, and monitorable.**

**PLAN & TRACK**

- Agile approach
- Sprint planning
- Sizing
- Task breakdown

**DEVELOP**

- Dev frameworks
- Integrated development environment (IDE)
- Code repository
- Checkout/pull request/commit
- Vagrant/Containers
- Sandbox
- Fast feedback

*Fast feedback with a Dev environment/framework allows for minimum viable product rework.*

**BUILD**

- Continuous build
- Security checks
- NFR validation
- Working code (master)
- Fast feedback
- Fast fix
- Image build

*Small tasks allow for rapid builds, minimizing merge conflicts and keeping master code functional at all times.*
*Rejected merges or non-functional commits get reworked.*

**TEST**

- Unit testing
- Integration testing
- Security testing
- Immutability testing
- License validation
- Common Vulnerabilities and Expo- sures (CVE) checks
- Validation against best practices
- Clean Code validation
- Human acceptance

*Automated testing is often part of the build process, so this isn't always a separate phase. A successful build is a result of passing all validation tests.*

*If a build or test fails, review the objections and determine the resolution – then return to the code phase*

**RELEASE**

- Code repository
- Container registry
- A/B build
- Release committed and tagged
- Release notes
- Release documentation
- Automated documentation
- Operational documentation
- Operational handover

**DEPLOY**

- Traffic/user migration
- A/B or canary testing
- Real user acceptance
- NFR validation
- Feature validation

*Testing can continue through the **Deploy** phase, especially when there is a change to end-user experience. This allows for final consumer feedback before committing to the full release.*

*If final user testing fails, a rollback release may be necessary*

*Check for approval, dynamically rewrite infra- structure as code templates*

**OPERATE**

- Operational processes
- DevOps ownership
- On-call rotas
- Documentation improvements
- Operational feedback

*At this point, rollback should not be an option (unless some other processes have failed), but the feedback and improvement process starts.*

**MONITOR & OPTIMIZE**

**Monitor**

- Reliability
- Record NFRs and provide feedback
- Continuous testing
- Continuous penetration testing

**Optimize**

- Automatic:
  - Application utilization pattern analysis (via machine learning) to determine optimization actions
  - Metadata tag updates, enabling workload self-awareness
  - Update of workload, enabling self-optimization

***Continuous optimization** eliminates risks and waste associated with manually selecting cloud resources for your apps. At this point, you need to feed back to the **Code** and **Build** phases, automating resource allocation for your applications so they perform as they should.*

**SCRUM & AGILE**

Azure DevOps [Boards]
(Primary)

**SOURCE CONTROL**

Azure DevOps [Repos]

**LANGUAGES**

JS SQL

**FRAMEWORKS**

React

**INTEGRATED DEVELOPMENT ENV.**

VS Professional [SQL]
VS Code [All other]

**CONTAINERS**

docker

**BUILD PIPELINES**

Azure DevOps [Pipelines]

**AUTOMATED CODE QUALITY AND SECURITY**

sonarcloud

**INFRASTRUCTURE AS CODE/DECLARATIVE LANGUAGE**

Azure Resource Manager

**TESTING**

Azure DevOps [Test Plan]

**CODE REPOSITORIES**

Azure DevOps [Repos]

**CONTAINER REGISTRIES**

Azure Registry

**PACKAGE MANAGEMENT**

Azure DevOps [Artifacts]

**DOCUMENTATION**

Azure DevOps [Wiki]

**KEY MANAGEMENT**

Azure Key Vault

**INFRA DEPLOYMENT & CONFIGURATION MANAGEMENT (VMs)**

Azure Automation

**SECURITY**

Azure Security Centre

**ORCHESTRATION**

Azure Container Service

**KUBERNETES ORCHESTRATION**

Azure AKS

**MONITORING**

Azure Monitor
Azure Log Analytics

**INCIDENT RESPONSE / ITSM**

servicenow

**RESILIENCY**

Azure Site Recovery (VMs)